SEARCHING FOR INCOMPLETE SELF ORTHOGONAL LATIN SQUARES
— A TARGETED AND PARALLEL APPROACH

By

MIKE FENTON ©

A THESIS

Presented to the Faculty of

Lakehead University

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Ruizhong Wei

Thunder Bay, Ontario

April, 2003

SEARCHING FOR INCOMPLETE SELF ORTHOGONAL LATIN SQUARES
— A TARGETED AND PARALLEL APPROACH

Mike Fenton, Msc.

Lakehead University, 2002-2003

Advisor: Ruizhong Wei

The primary purpose of this dissertation is in the search for new methods in which to search for Incomplete Self Orthogonal Latin Squares. As such a full understanding of the structures involved must be examined, starting from basic Latin Squares. The structures will be explained and built upon in order to cover Mutually Orthogonal Latin Squares, Frame Latin Squares and Self Orthogonal Latin Squares. In addition the related structure Orthogonal Arrays, will be explained as they relate to Incomplete Self Orthogonal Latin Squares.

This paper also dedicates time to explaining basic search methods and optimizations that can be done. The two search methods of focus are the backtracking algorithm and heuristic searches. In our final method the two will work together to achieve an improved result. The methods currently being used to search in parallel are also provided, along with the necessary backup to there structure.

The main research of this paper is focused on the search for Incomplete Self Orthogonal Squares. This is done by breaking down the problem into four separate areas of the square. By separating the blocks it enables us to work on a smaller problem while eliminating many incorrect solutions. The solution methodology is broken up into three steps and systematically solving the individual areas of the square.

By taking advantage of the properties of squares to constrain our search as much as possible we succeeded in reducing the total search time significantly. Unfortunately, even with our improvement in the overall search time, no open incomplete self orthogonal latin square problems could be solved. Full results and comparisons to existing methods are provided.

Acknowledgments

I must thank Dr. Ruizhong Wei who helped me tremendously through all stages of my graduate work and worked as my advisor to ensure the completion of this work.

I would also like to thank Dr. Maurice Benson and the Computer Science Department of Lakehead University who provided me support whenever it was needed throughout my time as a graduate student.

Personally I must thank everyone in my life for their continuing support, regardless of circumstances they have always provided me with all the support I need to accomplish my tasks in life.

Contents

Chapter 1

Introduction

The study of mathematics and combinatorial theory has existed for thousands of years. From the first time a secret message was created, there has been an interest in deciphering it. The relatively recent advent of computers created an increased growth within the field through necessity. Codes and secrets are a prevalent part of the electronic society and will continue to be for the forseeable future.

Computers have also provided mathematicians greater ability to solve complex problems that would have been previously impossible. As the search for larger and more complex problems continues, optimization and new methods for the computations come from necessity. Parallel computers are widely available in research conditions and are quickly becoming required tools.

Latin Squares are a long standing favorite game of mathematicians, which also hold interesting properties for those in the combinatorics field. The main purpose of this dissertation is to investigate a specific type of Latin Square known as Incomplete Self Orthogonal Latin Squares (ISOLS). The majority

of existence problems have been solved, as such, a direct approach is taken in an attempt to solve the remaining open problems.

When solving such a large problem, many related methods must be employed together in order to solve the problem. Backtracking through the search space greatly saves time, while using recursive methodology reduces memory usage. Adaptations of many existing algorithms and techniques need to be combined in order to efficiently search the problem space.

The problem space itself is sufficiently large that an exhaustive search without optimizations would not be realistic. Employing the specific algorithms improves the situation, however, a primary key is the use of the parallel architecture available. The widely available parallel libraries provides the necessary tools, however, the algorithms require very careful planning to ensure the desired results are achieved.

In the rest of this chapter, we briefly discuss the basic concepts of Latin Squares. In the coming chapters we will go into more detail regarding the specifics of Incomplete Self Orthogonal Latin Squares (ISOLS) as well as the necessary algorithms used in searching for them.

In Chapters 2, 3 and 4, the MOLS, SOLS as well as Frame SOLS and ISOLS will be explained in detail. While many theorems exist, we will focus specifically on those related to identity and existence properties. Application of the structures used can be found immediately following there definitions. In Chapters 5 and 6, we discuss the backtracking and searching algorithms in relation to large search space problems and parallel computer structures. In Chapter 7, we contrast other methods that are currently being employed to solve the problem and their basis of study. In Chapter 8 we will delve into the specific algorithms used in searching for the open cases from a basic

standard point and from enhancing the program through parallel computing. Finally in Chapter 9, we give a brief conclusion to the results of the testing.

1.1 History of Latin Squares

Latin Squares have been an area of interest to mathematicians. Both the standard Latin Square and the Magic Square are common puzzles that are considered pleasurable pastimes of the mathematically inclined. Latin Squares have a great deal more applications than that of simple puzzles. The concept itself was first started and named by Leonhard Euler in a paper written in 1782.[Euler]

Euler's interest is said to have started from a problem known as the thirty six officers. Is it possible to arrange thirty six officers, each having one of six different ranks and belonging to one of six different regiments, in a square matrix formation six by six, so that each row and each column shall contain just one officer of each rank and just one from each regiment?

This problem intrigued Euler for many years, and in the end he did not find a solution to the problem. He did surmise that there was no solution to Squares of size two and six but was never able to prove it. In fact, the problem remained unsolved until over one hundred years later in 1900 when G. Tarry proved it to be impossible. [Tarry] This, however, was not done using an acceptable proof of the reasoning but instead by exhaustively searching every possibility.

During his original definition of the Latin Square, Euler defined each officer as belonging to two distinct groups. One which represented the rank and one to represent the regiment. Originally, he used Latin and Greek

characters to represent each group and individual. In this system, every row and column needed to have all the Latin and the Greek symbols once. This system was referred to as a "Graeco-Latin Square". Euler realized that the first step in creating the Square was to place one set of the systems. He started with the Latin symbols, and the first step became a Latin Square. To further simplify the search, using the same symbols for both the rank and the regiment reduces the problem into a construction of orthogonal Latin Square pairs of side six.

This problem created the field of study on Latin Squares. While many of their applications apply to load balancing, communications networking and experimental design, it's origin lies in a recreational hobby for mathematicians.

1.2 Basic Properties of Latin Squares

The most direct definition of a Latin Square is the following

Definition 1.2.1. *A Latin square of side n is an $n \times n$ array in which each cell contains a single element from a set X of size n, such that each element occurs exactly once in each row and exactly once in each column*

Construction of a Latin Square with the above conditions is a relatively easy task. In relation to the thirty six officer problem this type of Latin Square is the correct solution to the first half of the problem.

Example 1.2.1 Latin Square of Side two

A simple square of side two

1	2
2	1

Example 1.2.2 Latin Square of Side six

A more complex Latin Square of side six similar to the thirty six officer problem

1	3	6	4	5	2
2	1	3	5	4	6
4	6	2	3	1	5
3	5	1	2	6	4
5	2	4	6	3	1
6	4	5	1	2	3

1.2.1 Operations on Latin Squares

A very important property of Latin Squares is that given a Square the rows and columns can be permutated to create new Squares. This property allows for the same data to be represented in new ways, yet, still retain the same properties. This will be shown to be to be very significant to solving problems in this paper.

Example 1.2.3 Properties of a Latin Square

1	2	3
3	1	2
2	3	1

This Square can be permutated into the following squares, however they remain isomorphic as the data in relation to all other data in the square remains unchanged.

Square 1

1	2	3
2	3	1
3	1	2

Square 2

2	1	3
3	2	1
1	3	2

Square 3

2	1	3
1	3	2
2	3	1

Square one shows an exchange of the second and third rows, Square two shows an exchange of the first and second columns while Square three shows an exchange of the symbol for one and the symbol for two. Any of the these methods create a new square and do not invalidate the Square.

The original square from Example 1.2.3 holds an important property in the definition of a Latin Square problem.

1.2.2 Standardized Latin Squares

Definition 1.2.2. *A Latin Square of side n (with the symbol set 1, 2, ..., n) is reduced or is in Standard Form if in both the first row and the first column the elements occur in their natural order. (1, 2, 3, 4... etc)*

By realizing that even while maintaining a standard form for the Latin Square it in no way reduces the generality of the solution, it can be employed to greatly reduce the search space when searching for a Latin Square.

Example 1.2.4 36 Officers Standardized

Returning to the thirty six officer problem the above Latin Square of side six can be Standardized to the following square

1	2	3	4	5	6
2	6	1	5	4	3
3	4	5	2	6	1
4	5	6	3	1	2
5	1	2	6	3	4
6	3	4	1	2	5

1.3 Orthogonal Latin Squares

Euler's primary focus was on that of the Orthogonal Latin Square. When the concept of Latin Squares was primarily a game for mathematicians, it was commonly referred to as a Magic Square. A Magic Square's primary principle is to have every row and every column add up to the same number. This is a necessary property for the existence of any Latin Square. Building upon the existence of a single Latin Square, it is a common situation in which two conditions apply.

In order to have a two conditional Latin Square, such as in the thirty six officer problem, we must be able to overlay the two Latin Squares such that the new Square satisfies a new property. No pairing of numbers can be repeated in the Square, and every pair of numbers must occur exactly once, leading to the definition.

Definition 1.3.1. *A pair of Latin squares $A = (a_{ij})$ and $B = (b_{ij})$ are orthogonal only under the condition that all the ordered pairs (a_{ij}, b_{ij}) are distinct for all i and j.*

In order to help visualize the overlaying of the two squares, we look at a simple example of a square of side 3.

Square 1

1	2	3
2	3	1
3	1	2

Square 2

2	1	3
3	2	1
1	3	2

Square 1 and 2 Overlayed

1	2	2	1	3	3
2	3	3	2	1	1
3	1	1	3	2	2

As can be clearly seen, when the two orthogonal squares are placed on top of one another there are no duplicated pairs. When searching for Orthogonal Latin Squares this is a vital property which must be taken into account.

Theorem 1.3.2. *[Colbourn] Orthogonality is symmetric. If a Latin Square L_1 is orthogonal to L_2, then L_2 is orthogonal to L_1.*

Proof. (Theorem 1.3.2) Suppose L_1 and L_2 are of side k and L_1 is orthogonal to L_2. Select j in the range $1 \leq j \leq k$. The positions where L_1 contains 1 include precisely one position in which L_2 contains j. Similarly there is precisely one position where L_1 contains 2 and L_2 contains j. Proceeding in this way we find all k positions in which L_2 contains j. L_1 contains 1, 2, ..., k in those positions, once each. As this is true for every j, L_2 is orthogonal to L_1. □

Finding Orthogonal Latin Squares can be a very daunting task. It is evident that with a Square of side two, it is impossible to find Orthogonal Latin Squares. In Euler's time, he was able to find a solution for a Square of side four as well as a large number of the odd numbers, Orthogonal Latin Squares with an odd side appear far less difficult to discover. Euler's inability to find a solution to the side six problem as well as ten and fourteen which he also investigated let to his famous conjecture.

Conjecture 1.3.3. *[Euler] No Orthogonal Latin Squares exist of side N when* $N \equiv 2(mod\,4)$

Euler's conjecture would be proven correct in 1900 by Tarry for side six, however, mathematicians were unsatisfied with his hand enumeration of the problem, and almost fifty years later in 1949, Bruck and Ryser found a solution to the problem by focusing on the existence of the projective plane.

Theorem 1.3.4. *[Bruck] (Bruck-Ryser) If* $N \equiv 1, 2(mod\,4)$*, then a necessary condition for the existence of a finite projective plane of order n is that integers x, y exist satisfying* $n = x^2 + y^2$.

From theorem 1.3.4, it can be observed seen that side six cannot exist as there is no way to square two integers to equal six. However, this did open up other solutions to exist, for instance side ten, can exist as its non-existence is not supported by theorem 1.3.4. Proof of the theorem is relatively complex but can be found in [Bruck]

In 1960 Bose, Shrikhande, and Parker were able to prove that except for the case of side six, Euler's original Theorem was incorrect and solutions found for side ten. With this discovery, research shifted from that of find-

ing the existence of Orthogonal Squares, to finding the number of Mutually Orthogonal Squares.

1.3.1 Transversals

A transversal in a Latin Square is a complete path through every row and column which only contains each of the symbols once. A partial transversal of length k is of the same concept yet is only of a portion k of the entire side. It has been demonstrated that some Latin Squares have no transversals while others have millions.

While a transversal may appear to be a simple concept it's application to more complex types of Latin Squares is important. Having a transversal can also provide information about related Squares.

Theorem 1.3.5. *[Colbourn] If a Latin Square L has a transversal, then any Latin Square isotopic to L also has a transversal.*

The number of transversals that exist can also play an important role of the existence of Orthogonal Latin Squares. This leads to two important Theorem's.

Theorem 1.3.6. *[Colbourn] Every Latin Square of even order has an even number of transversals. Every Latin Square of odd order has a transversal.*

Theorem 1.3.7. *[Colbourn] For a Latin Square L of side n to have an orthogonal Latin Square it must possess a set of n partial transversals.*

1.3.2 Diagonal Latin Squares

An important case of transversals occurs when considering Diagonal Latin Squares. In order for a diagonal Latin Square to exist it must contain com-

plete transversals along the diagonal.

Theorem 1.3.8. *[Colbourn] Let L be a Latin Square of side n. The Square L is a Diagonal Latin Square if both it's 0th right and (n-1)st left diagonals are transversals.*

Theorem 1.3.9. *[Colbourn] A Diagonal Latin Square of side n exists if and only if* $n > 3$

Example 1.3.1 Diagonal Latin Squares

[Colbourn] Squares of Side 4, 5, and 6.

0	2	3	1
1	3	2	0
2	0	1	3
3	1	0	2

1	3	4	0	2
3	2	0	1	4
0	4	3	2	1
2	0	1	4	3
4	1	2	3	0

1	2	3	4	5	0
0	5	4	3	2	1
2	4	0	1	3	5
3	1	5	2	0	4
5	3	1	0	4	2
4	0	2	5	1	3

Shifting back to the focus of this paper, the orthogonality of the Latin Squares, there exist some special cases where it is not possible to find an orthogonal square to a diagonal latin square.

Theorem 1.3.10. *[Hall] A pair of orthogonal diagonal Latin Squares of side n exist if and only if* $s \neq 2$, *3, or 6.*

Theorem 1.3.11. *[Hall] A set of three orthogonal diagonal Latin Squares of side n exist except when*

$$n \in \{2, 3, 4, 5, 6\}$$

and possibly when

$$n \in \{10, 14, 15, 18, 21, 22, 26, 30, 33, 34, 46\}$$

Chapter 2

Mutually Orthogonal Latin Squares

Building upon the concept of the Orthogonal Latin Squares, Mutually Orthogonal Latin Squares refer to groups in which all squares are orthogonal to one another.

Definition 2.0.12. *A set of Latin Squares $L_1, ..., L_m$ is mutually orthogonal, or a set of MOLS, if for every $1 \leq i < j \leq m$ are orthogonal.*

Existence of MOLS has been well researched and the following Theorem by Bose et al. stands true.

Theorem 2.0.13. *[Brouwer] There exist an MOLS(v) for any positive integer v, $v \neq 2, 6$.*

2.1 Existence Bounds on MOLS

The maximum number of Mutually Orthogonal Latin Squares for each case size has been researched thoroughly. In fact, full chapters of books have been dedicated to detailing the Squares, as well as the methods used to find some of the larger ones, see [Colbourn], [Abel], [Brouwer] . Focusing on strictly the maximum number of mutually orthogonal Latin Squares, the following theorem applies.

Theorem 2.1.1. *[Brouwer] A set of $k-1$ Mutually Orthogonal Latin Squares of side N is a complete set of MOLS. As such the maximum number of MOLS of side N is $k-1$.*

Proof. (Theorem 2.1.1) Proof based upon non-existence of k MOLS. Suppose $L_1, L_2, ..., L_k$ are pairwise orthogonal Latin Squares of side N, where k is equal to N. Suppose they are all standardized. Write a_i for the (2, 1) entry of L_i. If $A_i = 1$ then L_i has two entries equal to 1 in its first column, which is impossible. So the k numbers $a_1, a_2, ..., a_k$ are chosen from the k-1 numbers 2, 3, ..., k. By the pigeonhole principle $a_i = a_j$ for some i and $j, i \neq j$. This means that L_i has entry a_i in cells $(1, a_i)$ and (2, 1), and so does L_j. But orthogonality implies that, if L_i has the same entry in two given positions, then L_j has distinct entries in those two positions. A contradiction. □

Example 2.1.1 MOLS of Side 4

[Brouwer] A complete set of Mutually Orthogonal Latin Squares of side four.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

1	2	3	4
4	3	2	1
2	1	4	3
3	4	1	2

1	2	3	4
3	4	1	2
4	3	2	1
2	1	4	3

2.1.1 Finding N(n) for MOLS

Focusing on the maximum number of MOLS that can exist there are a number of Theorem's which define value for $N(n)$ as it applies to the side of the square.

Theorem 2.1.2. *[Brouwer] Using the principles of block design, if there is a* $(k^2, k^2+k, k+1, k, 1) - design$, *then* $N(k) = k - 1$.

Proof. Suppose a block design with the stated parameters exists. From Block Design, the set of blocks of the design can be partitioned into $k+1$ parallel classes of k blocks each; blocks in the same parallel class have no common element, while two blocks in different classes have one element in common. Suppose the parallel classes have been numbered 0, 1, ..., k in some order, and the blocks in class i also have been labelled $B_{i1}, B_{i2}, ..., B_{ik}$. Select two parallel classes for reference purposes, say class 0 and class k. Construct a square array from the i^{th} class, where $1 \leq i \leq k-1$, as follows. Find the point of intersection of B_{0x} with B_{ky}. The point will lie on precisely one line of class i. If it lies in B_{ir}, put r in position (x, y) of the array. Column y of the array will contain all the numbers of the lines in class i which contain a point of B_{ky}. Since the points of B_{ky} lie on one each of the lines in class i, the column will contain (1, 2, ..., k) in some order. A similar system applies to rows. Thus, the array is a Latin Square.

Write L_i and L_j for the Latin Squares obtained from parallel classes i and j respectively, where $i \neq j$. Those cells where L_i has entry r all come from points on block B_{ir}. The elements of B_{ir} consist of one element of B_{j1}, one from B_{j2}, ..., and one from B_{jk}. So the cells contain 1, 2, ..., k once each. So L_i is orthogonal to L_j. □

Theorem 2.1.3. *[Hall] For $k \geq 3$ and $k = p^\alpha$ where p is a prime number and α is a positive integer, there exists a set of $k-1$ orthogonal Latin Squares of order N, hence $N(k) = k - 1$*

Proof. We prove the theorem by showing a construction procedure for a set of k - 1 orthogonal Latin Squares. Let $b_1, b_2, b_3, ..., b_k$ denote the elements in the Galois field GF(p^α). We construct a set of $k-1$, $k \times k$ squares $A_1, A_2, ..., A_{n-1}$ with entries

$$\begin{aligned} a_{ij}^{(e)} &= b_e * b_i + b_j \\ i, j &= 1, 2, ..., n-1, n \\ e &= 1, 2, ..., n-1 \end{aligned}$$

We now notice that each of A_1, A_2, ..., A_{n-1} is a Latin Square. Suppose that there are two entries in the ith row of A_e which are the same; that is, $a_{ij}^{(e)} = a_{ik}^{(e)}$ for some j and k. This means that

$$\begin{aligned} b_e * b_i + b_j &= b_e * b_i + b_k \\ b_j &= b_k \\ j &= k \end{aligned}$$

Suppose that there are two entries in the ith column of A_e which are the same; that is, $a_{ji}^{(e)} = a_{ki}^{(e)}$ for some j and k. This means that

$$\begin{aligned} b_e * b_j + b_i &= b_e * b_k + b_i \\ b_e * b_j &= b_e * b_k \end{aligned}$$

Taking the multiplicative inverse of b_e

$$\begin{aligned} b_j &= b_k \\ j &= k \end{aligned}$$

We also notice that the set $A_1, A_2, ..., A_{n-1}$ is a set of orthogonal Latin Squares. Suppose that in the Latin Squares A_e and A_f, for some i, j, k, and l

$$\begin{aligned} a_{ij}^{(e)} &= a_{kl}^{(e)} \\ a_{ij}^{(f)} &= a_{kl}^{(f)} \end{aligned}$$

Then

$$b_e * b_i + b_j = b_e * b_k + b_l \tag{2.1}$$

and

$$b_f * b_i + b_j = b_f * b_k + b_l \tag{2.2}$$

Subtracting Equation 2.2 from Equation 2.1, we obtain.

$$b_e * b_i - b_f * b_i = b_e * b_k = b_f * b_k \tag{2.3}$$

Then

$$(b_e = b_f) * b_i = (b_e - b_f) * b_k \tag{2.4}$$

Since $b_e \neq b_f$ means that $b_e - b_f \neq b_n$. There is a multiplicative inverse of $b_e - b_f$ in the field. It follows that

$$\begin{aligned} b_i &= b_k \\ i &= k \end{aligned}$$

Equation 2.1 becomes

$$\begin{aligned} b_e * b_i + b_j &= b_e * b_i + b_l \\ b_j &= b_l \\ j &= l \end{aligned}$$

□

Theorem's 2.1.2 and 2.1.3 focus on isolating the complete MOLS, however, there are many cases where it is not possible to find the complete set. While there is currently no steadfast rule for calculating the number of MOLS that exist for a given side k, there are two theorems to assist in finding the bounds.

2.1.2 Minimum Bounds on MOLS

Theorem 2.1.4. *[Brouwer] If there exist n Mutually Orthogonal Latin Squares of side k_1 and n Mutually Orthogonal Latin Squares of side k_2, then there exist n Mutually Orthogonal Latin Squares of side k_1k_2. Hence,*

$$N(k_1k_2) \geq min(N(k_1), N(k_2)).$$

Proof. (Theorem 2.1.4) We take as the symbol set the ordered pairs of symbols. We form the order mn squares as m by m block structure matrices with each a block having size n by n. If the k^th square of the order n square contains the symbol a in location (R,C) and the k^th square of the order m square contains the symbol b in location (r,c), then the k^th square of the order mn square will contain the symbol ab in location (R_r, C_c). □

Theorem 2.1.5. *[Hall] Let $p_1^{\alpha_1} p_1^{\alpha_2} ... p_t^{\alpha_t}$ be the prime power decomposition of the positive integer n, where the p_i represents a positive integer. Let r denote the smallest of the t quantities $(p_1^{\alpha_1} - 1), (p_2^{\alpha_2} - 1), ..., p_t^{(\alpha_t)} - 1)$. Then there exists a set of r Mutually orthogonal Latin Squares of order n.*

Proof. By Theorem 2.1.3 there exists a set of $p_1^{\alpha_1} - 1$ orthogonal Latin Squares of order $p_1^{\alpha_1}$, a set of $p_2^{\alpha_2} - 1$ orthogonal Latin Squares of order $p_2^{\alpha_2}$, ..., and a set of $p_t^{\alpha_t} - 1$ orthogonal Latin Squares of order $p_t^{\alpha_t}$. Let us arbitrarily select r orthogonal Latin Squares from each of these sets. According to Theorem 2.1.4 they can be composed to yield a set of r orthogonal Latin Squares or order n. □

The creation of Mutually Orthogonal Latin Squares is an important endeavor in the field of combinatorics. It is a starting point in the understanding of building the more complex varieties of Latin Squares as well as other related structures. A more complex variety will be covered in chapter 4 and is the primary focus of this dissertation.

Chapter 3

SOLS and Orthogonal Arrays

Self Orthogonal Latin Squares is an important area of research which builds upon the basic concepts of Latin Squares as well the orthogonal principles. In order for a Latin Square to be Self Orthogonal it must be Orthogonal to its own transpose.

Self Orthogonal Latin Squares can be viewed as a a combination of the previous information regarding the Latin Squares. It must hold the basic properties of a Latin Square, while also adding at least one Mutually Orthogonal Mate. Once you combine all these related sets of properties together the constraints placed upon the existence become much more specific, as will be seen in Section 3.1.

As we move closer to the focus of this paper, the requirements on the existence of the square become increasingly specific. By understanding all conditions; both ensuing valid results as well as reducing the search space is possible.

3.1 Properties of SOLS

Definition 3.1.1. *A self-orthogonal Latin Square of order n, or SOLS(n), is a Latin Square of order n which is orthogonal to its transpose.*

Due to the specific nature of the relation of the square to itself, many of the previously covered topics hold specific application to SOLS. Referring back to Section 1.3.1, having a transversal is a necessary condition for an orthogonal mate to exist. The same is true for a self orthogonal Latin Square. The Square must contain at least one transversal. However in this case, that transversal must be it's main diagonal (right diagonal). A standardized SOLS has the numbers logically ordered along that transversal.

This fact also adds the interesting property that any self-orthogonal Latin Square can be changed to an idempotent latin square simply by renaming the symbols.

Theorem 3.1.2. *[Zhu] Self-Orthogonal Latin Squares exist for all orders of n when* $n \neq 2, 3, 6$.

Due to the wide acceptance and available of proof for theorem 3.1.2 it will not be provided here but can be located in [Brayton].

Example 3.1.1 SOLS

[Zhu] Self Orthogonal Latin Squares of side 4, 5, 7, 8, and 9.

1	3	4	2
4	2	1	3
2	4	3	1
3	1	2	4

1	5	4	3	2
3	2	1	5	4
5	4	3	2	1
2	1	5	4	3
4	3	2	1	5

1	7	6	5	4	3	2
3	2	1	7	6	5	4
5	4	3	2	1	7	6
7	6	5	4	3	2	1
2	1	7	6	5	4	3
4	3	2	1	7	6	5
6	5	4	3	2	1	7

1	4	7	6	3	5	8	2
7	2	1	3	6	8	5	4
6	8	3	1	7	2	4	5
8	6	5	4	2	7	1	3
4	1	2	8	5	3	6	7
2	7	4	5	8	6	3	1
5	3	8	2	4	1	7	6
3	5	6	7	1	4	2	8

1	5	9	6	2	4	8	7	3
8	2	6	9	7	3	5	1	4
2	1	3	7	9	8	4	6	5
7	3	2	4	8	9	1	5	6
6	8	4	3	5	1	9	2	7
3	7	1	5	4	6	2	9	8
9	4	8	2	6	5	7	3	1
4	9	5	1	3	7	6	8	2
5	6	7	8	1	2	3	4	9

While the Self Orthogonal Latin Square holds many features unique to itself, it can also be shown to have equivalent features to other problems. Understanding the relation to the other structures aids in the understanding of, and the search for SOLS.

Theorem 3.1.3. *[Zhu] A self orthogonal latin square of order n (SOLS(n)) is equivalent to.*

1. *An orthogonal array of strength two and index one, OA(4, n), which is invariant under the row permuation (12)(34).*

2. *A (2, 1, 3)-conjugate orthogonal Latin Square (or quasigroup) or order n, ie. a Latin Square which is orthogonal to its (2, 1, 3)-conjugate.*

3. *A spouse avoiding mixed doubles round robin tournament (SAMDRR) with n couples.*

Definition 3.1.4. *A spouse avoiding mixed doubles round robin tournament is a schedule of matches for n couples in tennis, such that*

1. *Husband and wife never appear in the same match either as partners or opponents.*

2. *Each pair of players of the same sex oppose each other exactly once.*

3. *Each pair of players of the opposite sex, not husband and wife play in exactly one match as partners and in exactly one match as opponents.*

An idempotent SOLS (A_{ij}) of order n can be used to schedule the $n(N-1)/2$ matches in the spouse avoid mixed doubles round robin tournament as follows: In the unique match in which Mr. i opposes Mr. j $(i \neq j)$, the partner of Mr. i is Mrs. a_{ij} (and the partner of Mr. j is Mrs. a_{ji}).

3.2 Basic Facts about SOLS

Having covered both the basic definition and the equivalence features of SOLS, we will now focus our attention on the properties of the Self-orthogonal Latin Squares themselves.

Definition 3.2.1. *[Zhu] Two self-orthogonal Latin Squares L and L' of order n are isotopic if there are bijections θ, ϕ, α from rows, columns and symbols of L to the rows, columns, and symbols, respectively, of L' that map L or its transpose to L'.*

Looking back to example 3.1.1 any SOLS of side four or five is isotopic to the squares presented.

Theorem 3.2.2. *[Zhu] If a SOLS(m) and a SOLS(n) exist, then a SOLS(mn) also exists.*

Proof of Theorem 3.2.2 can be obtained directly from the MOLS equivalent of the theorem, see Theorem 2.1.4 and Proof 2.1.2.

3.3 Orthogonal Arrays

Before moving on to the concept of partial Latin Squares, some time must be spent explaining a highly related concept which will aid us in our understanding of the algorithm used as the basis of this dissertation. We have already thoroughly explained the concept of Mutually Orthogonal Latin Squares in Chapter 2. A structure that holds the same properties as a set of MOLS is an Orthogonal Array. An Orthogonal Array is defined as:

Definition 3.3.1. *An orthogonal array OA(n, s, t) of order n and depth s with strength t is a matrix with s rows and n^2 columns with entries the numbers 1, ..., n. The structure must hold the property that in every $t \times n$ submatrix, every $t \times 1$ column vectors appears the same number of times.*

Relating the OA structure to a Mutually Orthogonal Latin Square, if we associate each row of the OA(n, s) with the n^2 cells of an $n \times n$ MOLS and

construct s matrices $A_1, A_2, ..., A_s$, using the ith row of OA(n, s) to fill in the cells of A_i in the order given by the association of column with cells. The orthogonality of any two matrices is precisely equivalent to the orthogonality of the rows. The same process can be reversed to create an Orthogonal Array from a set of MOLS.

The basic properties that hold true for a MOLS also hold true for an Orthogonal Array.

Theorem 3.3.2. *[Bierbrauer] Properties of an Orthogonal Array*

1. *Permutation of the rows or columns of an OA(n, s) produces an OA(n, s).*
2. *Substitution of the symbols or numbers in an OA(n, s) produces an OA(n, s).*
3. *Two orthogonal arrays that may be obtained from each other are called equivalent.*
4. *If a row is removed from an OA(n, s), the remaining array is an OA(n, s-1).*

Theorem 3.3.3. *[Hall] If there is an $OA(n_1, s)$ and an $OA(n_2, s)$, then there is an $OA(n_1 n_2, s)$.*

Proof. (Theorem 3.3.3)
Let OA(n_1, s) be the matrix

$$A = (A_{ij}), i = 1, ..., s, j = 1, ..., n_1^2,$$

and OA(n_2, s) be the matrix

$$B = (b_{ij}), i = 1, ..., s, j = 1, ..., n_2^2,$$

Form a new matrix

$$D = (d_{ij}), i = 1, ..., s, j = 1, ..., n_1^2 n_1^2$$

by replacing a_{ij} in A by the row vector

$$(b_{i1} + m_{ij}, b_{i2} + m_{ij}, ..., b_{in_2^2} + m_{ij}), m_{ij} = (a_{ij} - 1)n_2^2 for every i, j$$

As the numbers a_{ij} run from 1 to n_1 and the numbers b_ij from 1 to n_2, the numbers $b_{it} + m_{ij}$ run from 1 to $n_1 n_2$, whence every d_{ij} is one of the numbers 1, ..., $n_1 n_2$. Consider the hth row and the ith row of D and let u, v be any two numbers in the range 1, ..., $n_1 n_2$. Then we may write

$$u = u_1 + (u_2 - 1)n_2, v = v_1 + (v_2 - 1)n_2,$$

with $1 \leq u_1, v_1 \leq n_2, 1 \leq u_2, v_2 \leq n_1$ uniquely. In A, let us determine j as that column in which $a_{hj} = u_2, a_{ij} = v_2$. In B, let us determine t as that column in which $b_{ht} = u_1 and b_{it} = v_1$. Then in D, in column $g = t + n_2(j-1)$, we have

$$d_{hg} = b_{ht} + (a_{hj} - 1)n_2 = u_1 + (u_2 - 1)n_2 = u$$

and

$$d_{ig} = b_{it} + (a_{ij} - 1)n_2 = v_1 + (v_2 - 1)n_2 = v$$

This yields the orthogonality of the h^{th} and i^{th} rows of D and so proves that D is an orthogonal array. □

Theorem 3.3.4. *[Liu] Suppose q is a prime power and $2 \leq k \leq q$. Then there exists an OA(k, q).*

Proof. Let $a_1, ..., a_k$ be k distinct elements in F_q. Define two vectors in $(F_q)^k$ as follows:

$$v_1 = (1, .., 1) and$$

$$v_2 = (a_1, ..., a_k).$$

Now, define an array A, having rows indexed by $F_q x F_q$, where row (i, j) is the k-tuple $iv_1 + jv_2$.

We prove that A is an OA(k,q). Let $1 \leq c < d \leq k$ and let $x, y \in F_q$. We want to find the unique row (i, j) of A such that A((i, j), c) = x and A((i, j), d) = y. This gives us the following system of two equations in F_q, in the two unknowns i and j:

$$i + ja_c = x,$$

$$i + ja_d = y.$$

Subtracting the second equation from the first, we obtain

$$j(a_c - a_d) = x - y$$

Since $a_c - a_d \neq 0$, there exists a multiplicative inverse $(a_c - a_d)^{-1} \in F_q$. Then we have the following:

$$j = (a_c - a_d)^{-1}(x - y).$$

Back substituting, we can solve for i:

$$i = x - ja_c = x - a_c(a_c - a_d)^{-1}(x - y).$$

Hence, A is an OA(k, q) ∎

Theorem 3.3.5. *[Liu] Suppose q is a prime power. Then there exists an OA(q + 1, q).*

Proof for theorem 3.3.5 is omitted as it widely available and can be found in [Hall].

3.3.1 Application of Orthogonal Arrays

Orthogonal Arrays are a highly useful structure with many security applications within computers. We will briefly explain three cases where this structure can be of use. This material is presented as in the problem of focus, the creation of two orthogonal arrays is a necessary byproduct. This sub step of the problem we are solving has its own useful features.

Authentication Codes

Orthogonal Arrays are very closely related to Authentication codes as they hold an equality property between them. This equality can occur if, and only if, the authentication matrix is the transpose of an orthogonal array OA(2, k, l) and the authentication rules are employed using equal probability. Doing the same procedure in the reverse also results in the desired effect.

Orthogonal Arrays have a significant use in preventing spoofing attacks during authentication, however, the OA must be equivalent to the code as described above. Please see [Gopalakrishnan] for more information.

Equidistant Codes

Equidistant code is a special case of the standard code in which the distance between the keywords is equal and defined by the code.

The structure of the above code will form a natural OA of the same pattern. As well should it be an optimal equidistant(n, M, d;q) code corresponding to an affine resolvable 2-design, the columns will form an OA of order 2. For greater details on equidistant codes please see [Colbourn].

Constructions using packing designs

Packing designs are a combinatorial design used to construct frameproof codes and traceability schemes. Obtaining a packing structure from an orthogonal array is quite easy.

Lemma 3.3.6. *[Wei] If there is an OA(t, k, s) then there is a t-(ks, k, 1) packing design containing s^t blocks.*

Lemma 3.3.7. *[Wei] If q is a prime power and $t < q$, then there exists an OA(t, $q+1$, q) and also a t-$(q^2+q, q+1, 1)$ packing design with q^t blocks.*

The direct application to both frameproof codes and traceability schemes are large. For full details on both subjects see [Wei].

These brief explanations of the applications are intended to provide some insight into the application. An interested reader should refer to the sources mentioned in order to gain a full understanding.

With the basics of Self-orthogonal Latin Squares and Orthogonal Arrays presented, our attention turns to the more complicated matter of Frame SOLS and Incomplete SOLS. The material presented in this chapter provides the required background to understand the specific problem that this paper is focusing on.

Chapter 4

Frame SOLS and Incomplete SOLS

The overall purpose in this paper is to provide a new method in the search for Incomplete Self-orthogonal Latin Squares (ISOLS). The ISOLS is actually a special case of Frame Self Orthogonal Latin Square. This special subcategory of SOLS has an interesting property. They all contain a hole set. The hole set represents cells within the Latin Square which contain no value.

In order to work with a Latin Square which has holes, or is incomplete, we must update our definitions of Latin Squares to allow for these new types of Latin Squares.

4.1 Redefining Latin Squares

Our previous definition of a Latin Square from 1.2.1 states that every element in a set X must appear in each row and column. Obviously, if we add an empty space in a row or column, it is no longer possible to have every element

appear. We update the definition of a Latin Square by defining both a subsquare and a Holey Latin Square.

4.1.1 SubSquares of Latin Squares

Definition 4.1.1. *If in a Latin Square L of side n the k^2 cells defined by k rows and k columns form a Latin Square of side k it is a Latin Subsquare of L.*

For a portion of the square itself to be a subsquare it must, if viewed as a separate Square, be a proper Latin Square. It is important here, as well, to remember that the symbols from the Square do not necessarily need to be of a sequential order. The symbols themselves can always be interchanged to the appropriate values without changing the properties of the Square.

Theorem 4.1.2. *[Abel] A Latin Square of side n with a proper subsquare of side k exist if and only if $k \leq [\frac{n}{2}]$.*

Theorem 4.1.3. *[Abel] There exists a latin square of side n which contains Latin Subsquares of every side k, $k \leq [\frac{n}{2}]$, if and only if $1 \leq n \leq 7$ or $n = 9$ or 13*

Definition 4.1.4. *Let S be a set and $H = S_1, S_2, ..., S_k$ be a set of non-empty subsets of S. A holey or incomplete Latin square having hole set H is a $|S| \times |S|$ array, L, indexed by S, which holds the following properties:*

1. *Every cell of L is either empty or contains a symbol of S.*
2. *Every symbol of S occurs at most once in any row or column of L.*
3. *The sub-arrays $S_i \times S_i$ are empty for $1 \leq i \leq k$ (these sub-arrays are referred to as holes).*

4. *Symbol $x \in S$ occurs in row or column y if and only if $(x, y) \in (S \times S) \bigcup_{i=l}^{k} (S_i \times S_i)$.*

The most obvious change to the Definition occur in property one where each cell can now be an empty hole. Properties three and four simply define the holes themselves and the properties under which they occur.

4.1.2 Partitioned Incomplete Latin Squares

Definition 4.1.5. *A case of the Incomplete Latin Square in which multiple holes exist is referred to as a partitioned incomplete latin square (PILS) and follows the pattern*

$$PILS(n; b_1, b_2, ..., b_k) where b_1 + b_2 + ... + b_k = n$$

Example 4.1.1 Partially Incomplete Latin Squares
A PILS(7; 1, 1, 1, 2, 2).

	4	5	6	7	2	3
3		6	7	1	5	4
2	7		1	6	4	5
7	6	2			3	1
6	3	7			1	2
4	5	1	2	3		
5	1	4	3	2		

If there exists a valid $ILS(n; b_1, b_2, ..., b_k)$ it is always possible to complete the square by filling in the hole of size $b_i, 1 \leq i \leq k$, with a latin square of side b_i (on the symbols B_i). The existence of a Latin Square of side n containing a Latin Subsquare of side k is equivalent to an ILS(n;k).

Theorem 4.1.6. *[Brayton] A* $PILS(n; b_1, b_2, b_3)$ *exists if and only if* $b_1 = b_2 = b_3$. *As well, a* $PILS(n; b_1, b_2, b_3, b_4)$ *exists if and only if* $b_1 = b_2 = b_3$ *and* $1 \leq b_4 \leq 2b_1$.

While it is possible to find many Latin Squares with proper Subsquares, it is also important to realize that there are also many cases when it is not possible to create a Latin Square in which a proper Subsquare exists. If a proper Subsquare cannot be found in a particular Latin Square that square cannot be part of an Incomplete Latin Square.

Theorem 4.1.7. *[Abel] There exists a Latin Square of side n with no proper subsquares if* $n \neq 2^a 3^b$ *or if* $n = 3$, *9, 12, 16, 18, 27, 81, or 243.*

Example 4.1.2 Latin Square without a Subsquare

A Subsquare of side one is by definition always possible. Latin Squares without a Subsquare of side two become a more important focus.

A Latin Square of order 8 with no subsquares of side 2.

1	2	3	4	5	6	7	8
2	3	1	5	6	7	8	4
3	1	4	6	7	8	2	5
4	6	8	2	1	3	5	7
5	8	2	7	3	4	6	1
6	5	7	1	8	2	4	3
7	4	5	8	2	1	3	6
8	7	6	3	4	5	1	2

4.2 Incomplete MOLS

Our definition for an Incomplete Mutually Orthogonal Latin Square builds directly from our previous definitions of both Incomplete Latin Squares and Mutually Orthogonal Latin Squares.

Definition 4.2.1. *Two incomplete Latin Squares (ILS($n; b_1, b_2, ..., b_s$)) are orthogonal if upon superimposition all ordered pairs in $(B \times B)$ $\bigcup_{i=1}^{k}(B_i \times B_i)$ result. Two such squares are denoted IMOLS($n; b_1, b_2, ..., b_s$)*

Theorem 4.2.2. *[Abel] Existence of IMOLS: For $k \geq 1, N(n; k) \geq 2$ if and only if $n \geq 3k$, and (n, k) $\neq$ (6, 1).*

The concept of IMOLS is a relatively simple as it builds from other basic principles. The concept will be concluded with a simple example.

Example 4.2.1 IMOLS of side 6

A pair of Incomplete Mutually Orthogonal Latin Squares of side 6.

The Missing Subsquare is on the symbols 5 and 6.

5	6	3	4	1	2
2	1	6	5	3	4
6	5	1	2	4	3
4	3	5	6	2	1
1	4	2	3		
3	2	4	1		

1	2	5	6	3	4
6	5	1	2	4	3
4	3	6	5	1	2
5	6	4	3	2	1
2	4	3	1		
3	1	2	4		

It is important to note that the missing square which would contain only the symbols 5 and 6 can be placed into either Square in the pattern.

5	6
6	5

6	5
5	6

However it is not possible to complete both squares and have them maintain their orthogonality to each other.

For detailed information on the current status on known Incomplete Squares and the current bounds of the problem please see [Abel] II. Chapter 3 which includes all current information in remarkable detail.

4.3 Frame Self Orthogonal Latin Squares

We now start to combine the previous concepts into the area of focus for this paper. We begin with a definition of a Frame SOLS.

Definition 4.3.1. *A frame SOLS (or holey SOLS) is a self-orthogonal Latin Square of order n with n_i missing SubSquares (or SOLS) of order $h_i (1 \leq i \leq k)$, which are disjoint and spanning. (ie. $\Sigma_{1 \leq i \leq k} n_i h_i = n$). It is denoted by FSOLS($h_1^{n_1} ... h_k^{n_k}$), where $h_1^{n_1} ... h_k^{n_k}$ is the type of the FSOLS.*

Frame Self Orthogonal Latin Squares become clear with an example.

Example 4.3.1 Frame SOLS

[Colbourn] Frame Self Orthogonal Latin Squares. FSOLS(2^n) for n = 4 and 5.

		7	6	8	4	3	5
		5	7	3	8	4	6
6	8			7	1	5	2
8	5			2	7	6	1
3	7	1	8			2	4
7	4	8	2			1	3
5	6	2	1	4	3		
4	3	6	5	1	2		

		5	9	8	6	4	3	2	7
		8	4	7	9	2	5	6	3
9	5			6	1	0	8	7	4
4	8			0	7	9	1	5	6
7	9	1	6			8	2	3	0
8	6	7	0			3	9	1	2
3	4	9	1	2	8			0	5
5	2	0	8	9	3			4	1
6	3	4	7	1	2	5	0		
2	7	6	5	3	0	1	4		

Example 4.3.2 Advanced Frames SOLS

[Zhu] Two more examples of Frame SOLS with slightly more complex patterns.

FSOLS(2^n1^1) for n = 4

		6	8	9	7	3	5	4
		7	5	8	9	6	4	3
8	6			7	1	9	2	5
5	7			2	8	1	9	6
4	9	1	7			2	3	8
9	3	8	2			4	1	7
6	4	5	9	3	2			1
3	5	9	6	1	4			2
7	8	2	1	4	3	5	6	

FSOLS($1^6 3^1 2^1$)

	7	5	2	a	9	4	b	3	6	8
6		8	7	3	a	1	5	b	4	9
9	4		a	8	1	b	2	6	5	7
5	b	9		7	2	6	a	1	8	3
7	6	b	3		8	2	4	a	9	1
b	8	4	9	1		a	3	5	7	2
3	a	6	1	b	5				2	4
4	1	a	6	2	b				3	5
a	5	2	b	4	3				1	6
8	9	7	5	6	4	3	1	2		
2	3	1	8	9	7	5	6	4		

The symbols a and b are used to represent the numbers 10 and 11.

FSOLS($2^2 1^6$) Holds a different type of structure

	8		9	3	2	7	6	5	4
9		6		7	8	0	5	4	2
	7		4	5	6	9	8	1	3
4		8		0	2	5	9	7	6
6	2	1	7		9	8	3	0	5
7	6	4	8	2		3	1	9	0
8	9	5	0	1	4		2	3	7
5	4	3	6	9	0	1		2	8
3	0	9	5	6	7	2	4		1
1	5	7	2	8	3	4	0	6	

As demonstrated in the last example, the diversity of the Squares is quite vast.

Many complex patterns can be created using the pattern in the Frame SOLS definition. An important point, which we have already referred to, is that each missing Subsquare, or hole, is a Self-Orthogonal Latin Square. As such looking at Example 4.3.1 and 4.3.2. It can clearly be seen that a matching pair of symbols are missing from rows and columns where the Square has been removed. In the first three squares, the symbols 1 and 2 have been removed.

This property of Incomplete SOLS leads us to a direct link to Orthogonal Arrays. While the Orthogonal Array can be used to have a look at entire sets of MOLS at a time, the same principles apply on the smaller scale. Each set of rows which are missing a subsquare in a SOLS must be orthogonal to its matching columns. The two sets of data are linked at a fundamental level. This connection will be covered in greater detail during the presentation of the algorithm used in this paper.

The concept of Frames can also be applied to the previously covered MOLS. A frame MOLS is a more complex structure than the previously mentioned IMOLS, and the application to our problem is that it relates directly to ISOLS. If L_1 and L_2 form an FMOLS, such that L_2 is the transpose of L_2 then L_1 is a FSOLS. No further details will be provided on FMOLS as they do not aid us further in our understanding of the problem at hand.

Theorem 4.3.2. *[Zhu] Existence Properties of Frame Self Orthogonal Latin Squares.*

1. *If there exists a FSOLS($a^n b^1$), then $n \geq 1 + 2b/a$.*

2. *If $a \neq b$ and there exists an FSOLS($a^n b^1$), then $n \geq 4$.*

3. *A FSOLS(2^n) exists if and only if $n \geq 4$.*

4. *A FSOLS($2^n 1^1$) exists if and only if $n \geq 4$.*

5. *For $b \geq 0$ and $b \neq 2$, there exists an FSOLS($2^n b^1$) if and only if $n \geq 4$.*

6. *For any $u \geq 3$, a FSOLS($2^n u^1$) exists if and only if $n \geq u + 1$.*

4.4 Incomplete Self Orthogonal Latin Squares

The Incomplete Self Orthogonal Latin Square is the primary focus of this paper. The square itself is relatively simple to define however less difficult to discover.

Definition 4.4.1. *An Incomplete SOLS is a self-orthogonal Latin Square of order n, missing a SubSquare SOLS of order k, denoted by ISOLS(n, k).*

The ISOLS has a direct equivalent definition to the Frame SOLS.

$$ISOLS(n, k) \equiv FSOLS(1^{n-k} k^1).$$

And the two cases of ISOLS(n, 1) and ISOLS(n, 0) are equivalent to SOLS(n).

Example 4.4.1 ISOLS(7, 2)

[Zhu] Incomplete Self-orthogonal Latin Square of side (7, 2)

0	2	4	x	y	1	3
y	1	3	0	x	2	4
x	y	2	4	1	3	0
2	x	y	3	0	4	1
1	3	x	y	4	0	2
3	4	0	1	2		
4	0	1	2	3		

In this case the symbols x and y are used to comprise the SubSquare's missing symbols.

The largest possible missing block is defined as

If ISOLS(n, k) exists then $n \geq 3k + 1$

There cannot be a block missing larger than the above rational and still have a valid ISOLS.

Theorem 4.4.2. *[Zhu] There exists an ISOLS(n, k) for all values of n and k satisfying $n \geq 3k + 1$, except for $n = 6$ and $(n, k) = (8, 2)$, and possibly excepting $n = 3k + 2, k \in \{4, 6, 8, 10, 14, 16, 18, 20, 22, 26, 28, 32, 34, 46\}$.*

One update must be made to the above Theorem 4.4.2 recent research has provided a valid solution for $n = 3k + 2$ where $k = 4$ using an exhaustive search method, yielding ISOLS(14, 4). Building upon that solution it was found in [Bennett] that special constructions using Frame Self Orthogonal Latin Squares could be used to solve an additional 10 of the open cases using the new ISOLS(14,4) solution.

As a result of this recent research Theorem 4.4.2 can now be updated.

Theorem 4.4.3. *[Bennett] There exists an ISOLS(v, k) for all values of v and k satisfying $v \geq 3k + 1$, except for (v, k) = (6, 1), (8, 2) and possibly for $v = 3k + 2, k \in \{6, 8, 10\}$.*

By finding ISOLS(14,4), researchers also provided us with a new Theorem regarding ISOLS and FSOLS.

Theorem 4.4.4. *[Zhu] There exists an FSOLS($1^{10}4^1$) with a symmetric holey transversal with a hole of size four.*

Proof. Given a valid ISOLS(14,4), the ISOLS(14,4) will have a symmetric holey transversal consisting of cells (1, 6), (6, 1), (2, 7), (7, 2), (3, 5), (5, 3), (4, 8), (8, 4), (9, 10), (10, 9). □

In fact, the ISOLS(14,4) will serve as our basis of comparison in this paper in regards to the total search time and speed of the algorithms used.

Chapter 5

Search Algorithms

Finding a solution to a problem requires that a set of possible solutions be investigated. There are many unique ways to investigate a set of possible solutions, however the method used becomes increasingly important as the size of the possible solution set, or search space, grows. Optimization of the search becomes a key factor, as doing a full search of the space may not be practical.

There is a fundamental property required of any algorithm meant to investigate a search space, it must be exhaustive. Exhaustive, or exhaustiveness refers to the thoroughness of the search, if there is a solution, then the algorithm must find it. A simple method to accomplish this is to try every possible solution. This method works in a very simple problem, but is not feasible in a large scale problem.

Once the algorithm is assured of it's exhaustiveness, it's optimization comes into question. We want the problem to be solved as fast as possible, or to check the least number of invalid solutions. There are many methods which aid us in reducing the search space as we progress to eliminate unnecessary

calculations.

This chapter will focus our attention on two specific types of searches which were employed in the search for the missing ISOLS. The first is backtracking which both reduces our search space and guarantees our exhaustiveness. The second is heuristic which allows the use of intuitive control to sample the data and help to generate a solution.

5.1 Basic Search Background

This section will provide us with a comparison point by explaining a very simple search algorithm problem which will be used for the remainder of this chapter as an example.

Probably the most common problem in search algorithms is that of the knapsack problem. Essentially the goal is to place a number of items in a knapsack in an optimal way. The optimal way can be defined by the problem, whether it be, items worth the most, or the most items, or so on. Relating this to mathematics, we define a capacity for the knapsack and use relational values for the data involved.

For example, if you wanted to create an array of numbers so that adding exactly five numbers they equal a set value of one hundred, you would place a number in the solution set and then test to see if it helps, if it doesn't you move onto the next possible solution. It is important to realize that every possible solution is tried for the given set of data.

The problem space is defined as

Definition 5.1.1. *Knapsack problem set*

Profits: $p_0, p_1, p_2, ..., p_{n-1}$;

Weights: $w_0, w_1, w_2, ..., w_{n-1}$;

Capacity: M

Target of Search: an n-tuple $[x_0, ..., x_{n-1}] \in \{0,1\}^n$ *such that*

$$P = \sum_{i=0}^{n-1} p_i x_i$$

is maximized

Is optimized to some condition whether it be a target value, or optimal solutions.

A simple search for the knapsack problem would be a looping structure, which tries every possible solution and saves only the most successful. Running this however would take alot of time, and in many of the branches an invalid solution would still be searched through, as there is no compensation included for a bad path.

A bad path defines a branch of a search tree in which no valid solution can exist. In an ideal situation these branches will not be searched any further than is needed to determine that it is not a valid path.

5.2 Backtracking

Backtracking, as mentioned previously, is an effective method for performing an exhaustive search of a problem space. Rather than operate as a simple loop, backtracking is done using recursive methods. This allows the problem to be built logically one step at a time trying all feasible solutions. As such,

all feasible solutions are considered and the optimal solution will always be found, or the solution if only one exists.

Going back to the knapsack problem defined in Definition 5.1.1, it is easy to see that in order to try all possibilities, 2^n possible solutions must be considered. Generation of all the possibilities alone could take a long time. Using backtracking, it is easy to pick just the first and then the second possible choices. As it backtracks, it guarantees exhaustiveness. The only addition is a control value to monitor the overall status when looking for a specific or optimal solution. The control value is updated as improved solutions are located.

The addition of this control value allows for virtually any problem to be adapted to a backtracking search. Additional optimizations in the algorithm can be reached by a method called "pruning". This refers to reducing the total search space by ceasing to search a branch of the tree when the solution can not be advanced on that path. For instance, when attempting to place five numbers that add to one hundred, if ninety-nine is placed first and the smallest number is fifty, it must backtrack after trying all combinations for the second location. The third location will never be tried without a valid solution at the second location, and so on.

Improvements upon these simple concepts will be looked at for the remainder of this section.

5.2.1 Backtracking Algorithm

In the majority of optimization problems the solution can be seen as an array X, chosen from a finite possibility set, P. In the backtracking methodology, the values in x are defined one at a time as the search progresses. The

maximum depth of the tree will always be the total number of items in the array X.

At any given step we will have a partial solution stored in the array X. This is a constraint on the solutions remaining in this path of the problem space. Based upon these restrictions, the values that can be added to X, can be further subdivided from P, to a subset of P, we will refer to as a choice set C. It is in the creation of the choice set that pruning occurs. If C is the empty set then all subtrees that begin with that value starting at the current location will be pruned from consideration.

Algorithm 5.2.1: BACKTRACK(i)

global X, C_i
if X is a feasible solution
  then do any necessary computations
Compute C_i
for every $x \in C_I$
do { Append x_i to X
     { Backtrack(i+1)

Checking for a successful path, or solution, is done at the beginning of the algorithm and the algorithm continues ensuring that it will find the optimal solution. The computations may be saving the data or comparing it to the previously found results in the optimal solution search. The remainder of the algorithm is relatively straight forward as the computing of C varies on the problem definition as the recursion occurs.

Advanced methods for finding optimized data can be used to further

reduce the search space, however, in the program of existence we are dealing with, these methods are highly specific and will be discussed in detail when discussing the specifics of this application.

5.2.2 Special types of Backtracking

In this section we will briefly discuss numerous other special cases of backtracking applications which help in the understanding of the overall concept as well as in implementing minor optimizations used in our application.

Generating Cliques

Generating Cliques refers to the process of building smaller subsets within the larger set. The primary area of research is that of generating all cliques and maximal cliques. All cliques represents all subsets that exist without any repetition, while maximal cliques are the largest set that is possible. The longest, of course, being either the correct solution or the closest solution to the correct one. Naturally, the existence of the maximal clique will provide us with the solution to the existence problem itself, and is thus a highly useful search parameter.

The primary usage of cliques is in graph theory, however cliques are also useful for bounding purposes and analysis in other search problems.

Bounding Functions

Bounding functions are a simple idea from a theoretical viewpoint, however they can become very complex in application. It relies on the setting of

bounds upon the results at specific points within the search.

For example, if you are trying to find ten values that add to a specific number, if at location five, you are either too high or to low the problem discards this path.

Algorithm 5.2.2: BOUNDING(i)

external $Profit(), Bound()$
global $X, OptP, OptX, C$
if X is a feasible solution

then {
- $P \leftarrow Profit(X)$
- **if** $P > OptP$
  - **then** {
    - $OptP \leftarrow P$
    - $OptX \leftarrow X$

Compute C
$B \leftarrow Bound(X)$
for Every$X \in C$

do {
- **if** $B \leq OptP$
  - **then return**
- Append x to X
- $BOUNDING(i+1)$

The concept of an optimal value can be altered for use in other types of searches, in an existence search, it may be dependent on some specific pattern in the data which leads to the belief that it is not possible to find a solution. As the partial solution grows, the bounding functions parameters

as well as the target value must all change in order to make the best decisions regarding the pruning of the tree.

The concept of bounding is a relatively complex one, which requires careful planning to achieve the desired results. As such, a great deal of research has been done on the subject and many different algorithms exist based upon different principles, such as Greedy algorithms, Maximal trials, Clique Testing, and Sampling.

Branch and Bounding Functions

Branch and Bounding is primarily a new application of a bounding function. As can be seen in algorithm 5.2.3 it is built from a similar approach.

While the bounding function itself is used to prune the tree, this alters the order it is processed. Typically the recursive calls for the search are done in some predetermined order, whether it be numerically or some other method. Branch and Bounding instead makes decisions at each step to choose the solution most likely to yield the desired result and processes that solution path first.

This can be done for two different reasons. The first is to aid in finding the desired solution, if looking for a minimum or maximum bound, possible solutions can be discarded at a quicker rate with a tighter bound. Trying the one most likely to provide a new optimal value first can result in a lower average time on the search. The second is for cases where finding any solution is the key, and trying the best option should result in finding the final solution in minimal time.

This simple concept can be used for the majority of the search problems.

Algorithm 5.2.3: BRANCHBOUND(i)

external $Profit(), Bound()$

global C

if X is a solution

then {
- $P \leftarrow Profit(X)$
- **if** $P > OptP$
  - **then** {
    - $OptP \leftarrow P$
    - $OptX \leftarrow X$

Compute C

$count \leftarrow 0$

for Every$X \in C$

do {
- Append x to X
- $nextchoice[count] \leftarrow x$
- $nextbound[count] \leftarrow Bound(X)$
- Advance count

Sort *nextchoice* and *nextbound* such that *nextbound* is in descending order

for $j \leftarrow 0$ **to** $count - 1$

do {
- **if** $nextbound \leq OptP$
  - **then return**
- Append nextchoice to X
- $BrandBound(i + 1)$

The only downfall is the increase in processing time if the problem cannot be accurately analyzed to find an optimal path from the current state.

The next section focuses on a different nature of search which plays an important role in a finding a single solution to a problem. While the backtracking algorithm alone could find it, the time to process may be very large. This new type of search uses a decidedly different approach.

5.3 Heuristic Search

While the backtracking algorithm is an exhaustive search, the Heuristic search focus instead on finding a solution close to the optimal solution as quickly at possible. In a large problem, even finding one solution could require a great deal of time using the backtracking method to search the tree. Sometimes it is better just to find a solution.

Heuristic algorithms are usually a somewhat randomized algorithm that uses a method similar to trial and error in order to find a desired result. The search tries a pattern than then updates it as a whole attempting to reach the desired result. Updates are done through what is called a neighbourhood function. The search calls this function to update based upon the design strategy.

The neighbourhood of a value are the items in the solution that are either similar or close to the item in question. The use of a neighbourhood function allows for changes to be made across the set of data in order to achieve results faster than by using an exhaustive search.

Algorithm 5.3.1: HEURISTIC(C_{max})

external N, H_n, PChoose a feasible solution X
Set OptX to X
while $(c \leq c_{max})$

do
- $Y = H_n(X)$
- **if** $Y \neq Fail$
  - **then**
    - $X = Y$
    - **if** $P(X) > p(OptX)$
      - **then** $OptX = X$
- Advance c

return $(OptX)$

The following are search methods for use in a heuristic search

1. Find a feasible solution $Y \in N(X)$ such that P(Y) is maximized (return Fail if there are no feasible solutions in N(X)/X).
2. Find a feasible solution $Y \in N(X)$ such that P(Y) is maximized. If $P(Y) > P(X)$, then return Y, otherwise return Fail. (Steepest Ascent)
3. Find any feasible solution in $Y \in N(X)$.
4. Find any feasible solution in $Y \in N(X)$. If $P(Y) > P(X)$, then return Y, otherwise return Fail.

The first two methods would be primarily used in an exhaustive search, while last two are primarily used in a random search.

There are many different methods used in the neighbourhood functions to reduce the times involved in search as well as the accuracy of the search, which will be examined in brief.

5.3.1 Heuristic Search Methods

There are many effective methods for building a design strategy. This section will provide a brief look at some of the methods that exist. The methods are presented as a way to provide a basis for comparison to the methods employed in our search strategy.

Hill Climbing

Hill climbing is very direct method. If the new set is not better than the last, it fails. This method relies on the gradual improvement of the set. When doing an exhaustive style search using Hill Climbing, the focus is primarily on looking for the steepest ascent that is possible.

The Hill climbing method, unfortunately, is prone to becoming caught in a local maximum that it cannot get out of. If it takes a wrong path, there is no method to backtrack out of it using Heuristics alone.

Simulated annealing

Simulated Annealing provides us a method to escape from the local maximum mentioned above. This method relies on a new variable referred to as temperature, which allows for a replacement of the key variable given specific

condition and probability.

From a base point of view, it allows for jumps back through the choice of a randomized neighbourhood search. This method is able to avoid most local maximums.

Tabu Search

Tabu search provides another method of avoiding the local maximum. This time it is done by only concerning the decision of the next value by the best of the feasible solutions. This allows for a backtrack if there are no better moves to advance but there is no current solution.

The obvious problem with this method is avoiding the looping that would occur in such a system. The key is the control list TabuList. It is used to track the changes and any change that exists within the Tabu List is forbidden to be repeated for the a specified time frame.

Controlling the time does not prevent that the change may have future value, but it prevents against both becoming stuck in a local maximum and in an endless cycle.

Genetic Algorithms

Genetic Algorithms take a fundamentally different approach. While the other methods start with a single feasible solution and try to build a solution, the Genetic Algorithm starts with a population of many feasible solutions. By mutating the various solutions with each other it attempts to create new solutions that have the desired properties.

This can be a very complex pattern of steps in the combination, this method works based upon every two parents creating two children solutions which inherit certain properties of both parents.

Genetic algorithms in their natural form can be highly complex, however, the concepts they use can be very useful in manipulating data in other search problems. They can very quickly build many variations on a set of data.

For a much more in depth explanation for each of these cases as well as examples and application please see [Kreher]

These somewhat brief examinations of various search methods are presented to serve as an introduction to the specific methods we employed in searching for ISOLS. The background provided will allow the reader to fully understand the methods used and their optimizations in regards to the problem at hand.

Chapter 6

Parallel Algorithms

Parallel programming represents a tremendous opportunity for solving problems which otherwise may take an infeasible length of time to compute. Parallel machines have the unique ability of grouping processors to share data and share work which allows for the subdivision of large problems into manageable pieces.

Designing a parallel algorithm can be quite complex as it is critical to manage the data. Ideally, the data can be shared between the various processors, as well as be protected. Careful planning can ensure that both can happen, and waiting time can be reduced.

This chapter focuses on the approaches which can be taken in developing a parallel algorithm. It will examine the physical structures involved in the parallel machinery and draw comparisons to the previously covered algorithms such as Backtracking and Heuristics from a parallel viewpoint.

6.1 Basic Design and Structure

There are three different types of approaches that can be used to develop an algorithm for use in a parallel environment. Each are useful for solving different problems.

1. Modify an existing sequential algorithm by adapting the naturally parallelizable portions.

2. Design a completely unique parallel algorithm, which may have no direct sequential equivalent.

3. Run a simple sequential program on multiple processors at the same time with different inputs collecting the results.

All methods can be quite useful, and in complex problems, combinations of the methods may be the optimal solution.

When developing in parallel, it is important to remember that if 8 processors are needed to process the data, at some point that data must be recombined. This takes extra overhead, and care must be taken to manage the data during the process to make sure that each process only uses its own data and does not alter data needed by any other process. Typically large programs will assign a control processor which oversees the work of the other processors.

Each processor in a parallel array should be viewed as a node. Each node is assigned either a job or part of a job which it calculates and returns a result. The combined results are then processed to form the final result.

6.1.1 Constraints on Parallel Algorithms

By adding additional power, the complexity naturally increases. Here we will define rules that will aid us in better understanding the building of a parallel algorithm.

Multiple Instructions

Clearly a single processor can only execute a single instruction at any given time. A parallel processor, on the other hand, can execute up to its physical number of processors processes. These processes can be of different natures, as mentioned previously, they may all run the same instruction set, using different sets of data. A program of this type is referred to as a SIMD (single instruction, multiple data). The more complex variety where the processors divide the data as well as the job type are referred to as MIMD (multiple instruction, multiple data)

The majority of parallel algorithms are based upon the SIMD model. While multiple instructions can be used to improve performance, it is sometimes more useful to begin with a SIMD and exploit the nature of the problem.

Number of Processors

When dividing the work, the number of processors available can play an important role. Should the program request more processors than are available, the machine in most cases will spool your requests. This causes major slow downs as your program is always waiting for itself. Work should be dis-

tributed among the available processors to optimize the performance. If the given problem size is N and there are P processors each processors portion of the problem should be N/P.

Shared Memory

Parallel computers generally do not have independent memory for every processor. Primarily they use what is referred to as PRAM (parallel random access memory) that all processors share. As such, every processor has equal access to all memory locations. The machine itself on a lower level will prevent its own simultaneous reads or writes to the same location, however, if it is not accounted for in the program, unexpected results can occur.

One of the keys to working with in PRAM environment is the careful management of the resources. Should the data not be properly protected it can become corrupted and the data invalid. Avoiding this case is a necessary set in designing an algorithm using the shared memory model.

Messages

In more complex computations, it may be necessary for processors to interchange data one or more times during the calculations. In the PRAM model of parallel computing this does not occur as frequently as in the distributed model, however, it is still important to consider regardless of the memory model employed. Should a program need to interchange even a single variable it may be necessary to halt everything and wait.

When interchanging data, the data being passed is referred to as a mes-

sage. The sender makes a message and sends it to its recipient if they are ready to receive it it accepts it and continues. Otherwise it must wait until the other side is ready.

When designing an algorithm, it is important to realize that the time to compile, send, and acknowledge receipt of the message takes time. Even if no waiting is involved processing must occur, as such, data should be interchanged sparingly. Often it is more efficient to regenerate small parts of a problem rather than pass the data between processors repeatedly.

Input and Output

Gathering the necessary data into the proper memory locations can be an important step in finding a solution. As well, the program must have some method for outputting the data in order for it to be useful.

A generally accepted standard is to assume when designing the algorithm that the initial data is stored where it needs to be and that the central command of the program ensures that as well as handles all output. While this portion is not itself done in parallel, it does store the data into parallel memory, if applicable, and writes the combined results of the parallel processors. In larger sets of data a parallel generation or reading method may be used, however, for our purposes we will not require that methodology.

Having covered the basic design parameters and the physical limitations we now move onto the software aspect of the design. Parallel programming has two main systems that are used for it's handling. OpenMP and MPI, OpenMP is built directly into most parallel systems, while MPI is done via message passing and library usage.

6.2 Parallel Libraries

Parallel algorithms can be larger divided into two types of approaches dependant on the problem. This section will explain the two types and then explain the methods used for each.

6.2.1 Fine and Coarse Grain

The grain of the problem refers to the detail required for it's parallel processing. For problems which must exchange a large amount of data, it is better designed using a fine grain approach, where problems in which each processor only needs to interchange data a select number of times in the entire running is a coarse grain algorithm.

Coarse grain problems are typically ones that can be run almost independently, possibly a one or two step parallel program. This type of parallel process would have sub loops and a fairly complex structure itself. A coarse grain algorithm is best support by the MPI library which will be covered next.

Fine grain problems are ones that utilize very small chunks of data in the processing. There may be hundreds of parallel steps and the data is usually highly dependent. Each threads typically runs a very small set of code and returns, in fine grain problems the task is to speed up existing structures by solving them in parallel. The fine grain is very useful for specific problems which have many restrictions on them, and is supported by OpenMP in most parallel machines.

6.2.2 Message Passing Interface

MPI is the acronym for Message Passing Interface. It is a set of library functions which facilitate passing of messages to and from the various processors. This of course requires careful planning to properly subdivide the algorithm so that it is time and processor efficient to pass the messages. This type of interface can be used with either the parallel model of memory or with a distributed memory design.

MPI is a very powerful system with many uses, it's primary design however is for the coarse grain of problem which is not our primary focus, as our problem due to its highly specific constraints is suited well to the fine grain algorithm approach.

6.2.3 OpenMP

OpenMP is built into the majority of parallel machines due to its wide acceptance and ease of use. It relies on fork and join parallel by taking a time consuming portion of the problem and dividing it into smaller chunks then recombining it afterwards. Due to the integrated nature there is no need for message passing or the equivalent as it allocates processors based on needs, up to the maximum specified.

The application itself is done simply as pragmas within the source code instructing the system to handle itself in a parallel method. Converting a sequential program to a parallel program is a very simple job. The design problems occur when it is necessary to optimize the algorithm.

OpenMP should be viewed as a system which creates parallel sections. Each section is responsible for an equal portion of a small job. Due to the

speed at which OpenMP can create the new threads and recombine them, it is effective to use in a fine grain problem. OpenMP has built in definitions which allow for the easy division of many types of loops, as well as, allowing the user to specify the recombination rules of the problem.

Due to our applications needs, OpenMP will be the approach of choice. The specific approaches used in solving our problem will be covered in chapter 8

6.2.4 Deadlocking

Deadlocking is a very important concern when it comes to parallel programming in general. Anytime two processors want to read or write from the same information at the same time they begin a race condition and the result can vary. This is usually solved by adding a locking mechanism. Unfortunately if not done carefully, you can have two processors waiting for each other to finish.

This is a much greater problem when using MPI as the messages themselves can become stuck while they are trying to interchange the data. However, in complex problems using OpenMP the same can occur. If attempting to solve a problem with many conditions, it may be necessary to lock not just the individual data that is being either read or written, but a section to make sure that none of the data that you are working with is replaced.

An easy way to visualize this is to think of a shopping cart. If you are trying to make a cake, you've got one person putting cake ingredients in the cart, and another taking them out, you still don't get a cake. While they may not being doing the same job at the same time, or in parallel thinking, even aware of what the other's job or actions are, the final result should be

a cake.

Avoiding these situations all comes down to careful planning. It should not be possible for two processors to be competing with each other and they should properly protect the necessary data to avoid unexpected results or tampering with the information.

6.3 Backtracking

Referring back to the backtracking algorithm presented in 5.2.1, it is evident that adding parallelism to this algorithm would not work. Recursive solutions cannot be done in parallel, as they would continually be requesting new processors. The parallel approach for this case would be a fine grain algorithm in which the processing is sped up.

The generation of new solutions, especially when there are many constraints can be done in parallel any processing of that data which needs to be done. This can be a common occurrence when handling highly restricted sets of data.

This may seem to be of little improvement, however, when actual running time is considered, we see a great improvement. Anytime we have a large amount of processing we split it, as mentioned above, it will reduce that sections processing time. The recursion set is a very small step, only to have the larger chunk of processing done in parallel.

6.3.1 Branch and Bounding

The more complex method of branch and bounding that was covered in 5.2.3 does give us a little more complex example to which explain the workings of

the parallel methods.

Algorithm 6.3.1: PARALLELBRANCHBOUNDSECTION1(i)

if X is a solution

then
- $P \leftarrow ComputeinParalel\ (Profit(X))$
- **if** $P > OptP$
  - **then**
    - Compute in Parallel $OptP \leftarrow P$
    - $OptX \leftarrow X$

Compute C

$count \leftarrow 0$

Looking at the opening portion of this algorithm it can be clearly seen that similar to in the backtracking algorithm that the comparison and calculations of profit can be done in parallel, but only one processor can be used for the remainder. Doing simple comparisons and assigning values to variables is a single processor task.

We now take a look at the second portion of the algorithm which is able to provide us with a greater level of parallel involvement.

Algorithm 6.3.2: PARALLELBRANCHBOUNDSECTION2(i)

for Every$X \in C$

do {
- Append x to X
- $nextchoice[count] \leftarrow x$
- $nextbound[count] \leftarrow Bound(X)$
- Advance count

Sort *nextchoice* and *nextbound* such that *nextbound* is in descending order

As mentioned, this section provides us with many opportunities to make use of our parallel architecture. Parallel sorting is a widely analyzed task, and can be accomplished relatively easily. The ideal approach in altering the do loop would be to have each processor run the Bound function on a set of the X data. If there are more sets then the processors will simply divide the work evenly, which the for structure in OpenMP will do automatically. This will fill both nextchoice and nextbound logically. Count should be naturally maintained as part of the for loop.

As mentioned previously there are many algorithms for parallel sorting which will not be covered in detail here please see [Berman] for details on parallel sorting.

The optimizations done for the branch and bound algorithm give the potential to greatly reduce the processing time, and are relatively simple to accomplish.

6.4 Heuristics

Parallelizing the heuristic algorithm can vary a great deal. The core processing in the heuristic algorithm can be isolated in the neighbourhood function. For the majority of neighbourhood functions adding parallelism should be rather simple as they work on sets of data. Simply dividing the data into the logical segments is usually successful.

Dividing data does not need to be fully contained and in a situation like a neighbourhood function, in most cases, cannot be. Due to the neighbourhood function being based upon groups of data rather than on point data, it may be necessary to pass more information than will be updated in order to achieve the desired result. The function will only need the existing data so no interchanging while processing is needed. However, it will need a segment on points on either side of its segment to fully calculate the problem.

Algorithm 6.4.1: PARALLELNEIGHBOURHOOD(i)

For a simple problem such as the average of a line of points.

Neighbourhood function $N = \frac{x-1+x+1}{2}$

If this was being done in parallel. In order to process segment

$$X_i, ..., X_{i+n}$$

then

$$X_{i-1}, ..., X_{i+n+1}$$

would need to be passed to each processor with of course a special condition built in for the end points.

The remainder of the Heuristic algorithm is processed just as the Backtrack Algorithm would be, even though there is no recursion. Each set is completely dependent on the last so there is no usefulness in having one process advance without the others.

Chapter 7

General Problem Solvers and SATO

It is important when developing a new algorithm, or solution to a problem to be aware of what is currently being done. The majority of research and focus is currently being placed on general problem solvers. A universal program is written which, given a problem, is expected to be able to find a result. This type of approach has been the dominant method for finding Latin Squares since 1990.

Out of the general solvers currently in existence, SATO (Satisfiability Testing Optimized) is the most efficient. It has solved many open problems, and is constantly being improved upon. SATO has moved the field of Latin Squares forward greatly by its approach and improvements made.

This chapter will explore the methods used in SATO and general solvers as a whole. We cover this material so as to better understand the existing methodologies being used and for later comparison of results.

7.1 Basics

As mentioned previously, the model generator has become the method of choice for solving open Latin Square problems. The approach taken to solving the problems must focus on two areas.

1. Developing an efficient model generation program

2. Provide efficient program specification to the program

The first is clearly a large task. However, most approaches are quite simple. By leaving a large portion of the optimization to the specification used, it can simplify the program as well as focus ones attention. The basic approach taken in programming a general solver is to load in a set of conditions and search through the possibilities for ones that satisfy the constraints. Clearly, the greater the number of constraints and the more organized they are, the better the results.

SATO goes a step beyond the basic in their general solver approach by including many latin squares optimization techniques in the code itself. These allow for new conditions and improvements to handling latin squares to, in a sense, be hardwired into the search for the specified problem.

The second is a very important situation because it allows for the solving of a huge variety of problems, and can vary greatly. While the program does define the format and methods used to specify the problems, the specification itself has proved to be an area where tremendous gains can be achieved through optimization.

A great deal of research has been done over the past few years regarding this optimization for various problems. Some solutions include adding extra

constraints, using cyclic group construction and checking for extra isomorphic cases.

SATO itself is a model generator based upon the Davis-Putnam algorithm for propositional clauses. SATO was largely developed to attack the open latin squares problems. Full details can be found on the SATO web site, as well as full source code is available on the SATO website.

http://www.cs.uiowa.edu/~hzhang/sato.html

The remainder of this chapter will focus on the specification of the latin squares and some simple optimizations. The approach taken to explain the system will be that of a general overview, for a full study please see, [Zhang], [Zhang 2] and their sources. The goal is to simply provide enough information under which to make a fair comparison of the results.

7.2 Problem Specification

The building of a specification begins with simple definitions and builds upon those until the problem is fully designed. The first step of designing a rule set is transferring the visualized Latin Square, or other problem into a Quasi-group of possible values at any given point.

To begin any latin square problem we must define the orthogonality principles of the problem. These are a general definition that can be used for virtually all problems.

$$QG0 : (x * y = z * w \wedge y * x = w * z) \Rightarrow (x = z \wedge y = w)$$

$$QG1 : (x * y = z * w \wedge u * y = x \wedge u * w = z) \Rightarrow (x = z \wedge y = w)$$

$$QG2 : (x * y = z * w \wedge y * u = x \wedge w * u = z) \Rightarrow (x = z \wedge y = w)$$

These three rules define the orthogonal conjugate constraints on any latin square. Even though these are very well defined starting conditions, they still remain as quite a difficult model for generation problems. Even just the existence of QG2(10) has yet to be determined.

These are of course not the simplest of the building blocks, however they do contain the more primary cancellation and closure properties within the orthogonal definition.

By constructing many sets of these rules it is possible to determine any applicable existence problem. Developing the conditions can, at times, be tedious and inefficient.

7.3 Example of SATO for Holey Latin Squares

Without getting into an extreme amount of detail, we will now cover the process of building a system using SATO to solve for holey Latin Squares. We will simply go over the sets involved, but because of the incredibly large number of rules needed, it will not be a complete working example.

To begin, we assume that the standard definitions already exist. Next we must define a binary relation, which we will call $same_hole$, on Latin Square $(S, *)$. The relation $same_hole(x, y)$ will return true under the condition that both x and y exist in the same hole.

With this relation defined, the constraints must be added to make sure of this function. For any constraint $a * b = c$ $!same - hole(a, b)$, $!same_hole(a, c)$ and $!(same_hole(b, c)$. This method on its own is very tedious and for a square with n free variables, $n(n-1)/2$ occurrences of $same_hole$ is required.

To optimize the method we take the following steps.

1. Treat each idempotent Latin Square of order v as a frame Latin Square of type 1^v.

2. Generate the negative unit clauses such as $x*y \neq z$ whenever $same_hole(x, y)$, $same_hole(x, z)$ or $same_hole(y, z)$ is true.

3. Attach the condition $!same_hole(x, y)$ to the closure properties.

4. In none closure properties, Attach the negative $same_hole$ to the clause if the positive literal $x * y = z$ does not have a negative literal as well.

Including all the necessary conditions and applying these rules results in the most optimized method for solving the incomplete latin square. Clearly organizing all the clauses can be tedious however when dealing with highly complex problems that is to be expected. Using the method overviewed above the creator of SATO was able to solve the problem for ISOLS(14,4) which as previously mentioned will serve as our comparison point for algorithm speed.

7.4 Other Optimization Techniques

This section will briefly cover some additional techniques used in optimizing the search parameters for use in a general solver environment. Understanding all the methods that are used to approach the problem leads us to a more complete understanding of the problem, as well as, provides additional methods to optimize our existing solution method.

7.4.1 Conjugate-orthogonalities

Referring back to QG0-QG2 defined in section [Zhang] we can see the basic definition of the orthogonality. To convert those two clauses into the propositional equivalents for a Latin Square of order v, requires $2v^6$ clauses. In a Latin Square due to cancellation, QG1 or QG2 are sufficient, which reduce the complexity by half.

An additional method used is the grouping of variables. By creating a new predicate, accepting three variables, we reduce the total free variables from 6 to 4 reducing the number of clauses from v^6 down to v^4.

These changes in semantics greatly improve the running time. In more general terms, it creates awareness of double searching for matching pairs. This case should not occur in an optimized search as if it fails or passes one of the conditions, the other must follow suit.

7.4.2 Isomorphism elimination

While Isomorphism elimination can be a very involved method to implement, the concept is very important. For any given latin square, there are many different patterns that can be created from it using Isomorphic properties. By simultaneously exchanging row and columns. By excluding these isomorphic cases, the total search space can be greatly reduced.

A common method for finding if this case will occur is called Least Number Heuristic (LNH) used in FALCON and SEM. It works in the early stages of the seach when many of the value names are symmetric and eliminates at an early stage. Using this method however, requires special constructions within the general solver and may not be possible in many cases. This,

however, remains an important method when searching for Latin Squares.

While SATO and other general solvers vary greatly from the methods being used in this paper, it is important to realize that both have a great deal in common. The general solver is a great tool for trying many problems. However, the specific approach used in this paper can prove to be helpful under circumstances when the problem grows to large to be feasible with the general solver.

The methods used in SATO influenced some of the choices made when developing the specific algorithm used by the author to develop a system to solve Incomplete Self Orthogonal Latin Squares. Full comparisons of SATO to the targeted approach can be found in chapter 9

Chapter 8

Searching for ISOLS

The focus of this paper comes from one relatively straightforward theorem. The problem of Incomplete Self Orthogonal Latin Squares has almost been solved completely. The following theorem describes the current level of research in this field.

Theorem 8.0.1. *There exists an ISOLS(v, n) for all values of v and n, satisfying* $v \geq 3n + 1$, *except for (v, n) = (6, 1)(8,2) and possibly for* $v = 3n + 2, n \in 6, 8, 10$.

Clearly this theorem leaves much open to investigation. Possibly is a questionable wording for a construct such as a theorem, yet nonetheless does accurately describe the situation we begin with.

Our focus was to develop an efficient way to search for the three open cases of ISOLS, namely (20, 6), (26, 8) and (32, 10). To that end we will employ a variety of measures to attempt to solve the problem.

This chapter will begin with a basic breakdown of the steps taken in solving the problem, and will follow up with the algorithms used at each

stage. When applicable, the parallel algorithms used will be explained as a subsection to the portion of the algorithm to which it applies.

8.1 Dividing and Conquering the problem

We begin by breaking down the problem. A finished Incomplete Self Orthogonal Latin Square can be viewed as four separate portions. We begin with a sample square which is full, with the exception of it's hole in the bottom right corner. This state is our finished Square.

The next step, when decomposing the problem, is to eliminate sets of values from the Square. Before doing this we must define the regions within the Square. We start by labelling the hole, as section D. The hole will always be square in shape. The columns and rows directly above and to the left of the hole will be sections B and C respectively. The large remaining section will be section A. While sections B and C may be squares, they do not need to be, however sections A and D must be a squares.

Visually we have

$$P = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array}$$

Where B is an $(2n+2) \times n$ rectangle and C is a $n \times (2n+2)(2n+2) \times n$ rectangle where n is the side of the hole. The rectangles which comprise sections B and C must be Orthogonal Arrays or OA as presented in chapter 3

At the same time, we will also divide the symbols used into two groups. One group will be from an extended set (E), we will use greek symbols for it,

the other will be the standard set (S), which will be represented numerically. The extended set represents the values which "should" be contained within the hole of section D.

The standard set is all values which would not exist in D, making it the larger of the two sets. By way of the property of Latin Squares, the hole D, must be a latin square. Therefore, it must be assumed that every value from our extended set appears in each row and each column exactly once. With that being the case, no symbol from the extended set can appear in the adjoining sections B and C.

Now that we have the Square subdivided, we will cover the steps involved in finding a solution.

8.1.1 Building Sections B and C

In the first stage of our problem, we focus on sections B and C, which will only have the standard symbols. This step is done first as it has a tighter constraint than attempting to solve section A from the beginning. The constraint comes from the fact that the Square must be self-orthogonal, so we can look at the pairings of values as we build to maintain our self-orthogonality.

This is done by viewing the superpositions of B with C^T. We may only place one set of points, one point in B and one point in C, at a time. The set of points is only valid if the unordered pair being placed does not already exist between B and C.

In any Square, the superposition of B and C^T will produce $n(2n+2)$ distinct unordered pairs on the standard symbol set, $Z_{2n+2} \times Z_{2n+2}$. The total number of unordered pairs from the standard symbol set is $(\frac{2n+2}{2})$ leaving a remaining $n+1$ unordered pairs which must exist in the superposition of A

and A^T.

8.1.2 Placing the Missing Pairs

The next step is to place the remaining unordered pairs. This, of course, is done in section A of the square. This step is done by considering section A and its transpose. All points must be placed within the square in valid locations for this section to be successful. Any square which can have all points placed, thus completing the set of unordered pairs on $Z_{2n+2} \times Z_{2n+2}$, will be referred to as an extendable solution.

Having placed all missing pairs, we are now ready to try to place the remaining points.

8.1.3 Finishing the Square

The final step is to place the set of points that corresponds to the overlaying of the two value sets. None of these pairs have been used and the extended symbol set has no restrictions on it in section A. In order to reduce the search time, we want to test against restrictions to reduce the options available.

The standard symbols that must be placed in A have existing constraints, from the predefined main diagonal, which is in standard form, as well as sections B and C, and from the previous step of placing the missing pairs of values. By the definition of a Latin Square we know that each symbol from the standard set must appear in every row and every column of A exactly once.

We exploit this principle by attempting to place each value from the standard set in each column, and should it be able to place the value in

every column, by definition, every row will also have the symbol placed. If the Square is able to accept all values from the standard set, it is a very simple process to place the matching extended set symbols making sure that each pairing only occurs once.

This results in our having found an Incomplete Self Orthogonal Latin Square. Each algorithm will now be explained in further detail, as well as optimization techniques used to attempt to find a solution for the larger problems when an exhaustive search is not feasible.

8.2 Building Sections B and C

The algorithm for building sections B and C in our application is done recursively and requires two functions which will be explained now as they will be referred to in each of the coming sections.

The first is CheckMainValid(), which searches through the row and column that, corresponding with the specified point, checking for the number requested. The function returns TRUE if the point is not found and FALSE if it is found. The second is very similar, CheckSecondValid(), which checks the matching pair, first that the location is valid by running CheckMainValid() on the second point and then by cross referencing the point against an array holding used pair data.

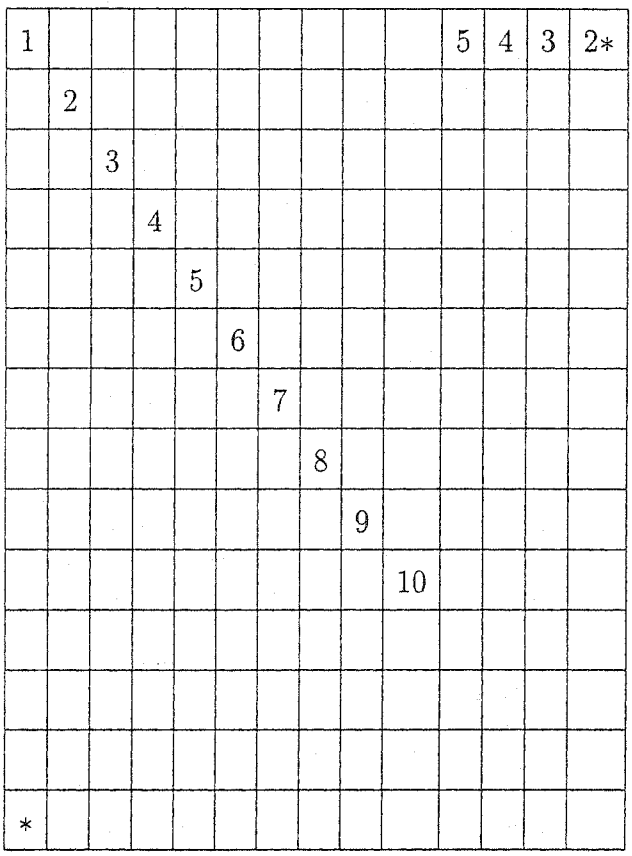
The used pair data is a binary array keeping track of which pairs of unordered points have been placed. Both the (x, y) and (y, x) coordinates are updated to ensure that regardless of the order of the points the data is found correctly when checked.

Before moving on to the algorithm itself, the initial conditions placed

upon the latin square should be explained. The algorithm runs in the memory space of a single Latin Square of the side requested to be solved. This square upon initialization is set to standard form. The main diagonal is in numerical order, guaranteeing the pairs, (1, 1), (2, 2), ..., (n, n) will only occur once and are present. As well, in a increasing order moving from right to left beginning at the first valid number, 2 in the standard form, the top row of B is labelled sequentially.

Example 8.2.1 A Blank Latin Square

The starting point for solving a ISOLS(14,4)



The same pattern applies to a Latin Square of any size.

The processing of the latin square begins in the top right corner, where

the two is located denoted in Example 8.2.1 by the asterisk, it then places these locations by matching values in the bottom left corner also denoted by an asterisk. Below is the basic algorithm used.

Breaking down the algorithm, it is not incredibly complex in it's basic form. The first step is to find a symbol which can be placed in section B. Once that is found, it attempts to place a symbol in the matching location in section C. The check for a valid symbol is done using CheckMainValid() and CheckSecondValid() respectively. This is done as a loop ensuring that every number is tried. When the recursive step backtracks, it continues at the point symbol it tried last and advances to the next symbol. By maintaining the recursive line, we guarantee an exhaustive search.

When the BuildSectionsBC algorithm returns, it is important to note that the value being tried in the LatinSquare must be reset to zero, as well as the UsedPair set, is marked as unused. The subloop allows the program to try every combination of Second for each Main as it proceeds.

Running the above algorithm will correctly yield all segments B and C that exist for the specified problem size. Now we will focus on the parallel enhancements which can be made to the algorithm.

8.2.1 Parallel Building of B and C

This section will cover not only the parallel used, but another method attempted for the discovery of the (32,10) sections B and C, which were more obscure due to the size of the problem.

To begin, at the simplest level, we began by parallelizing the CheckMainValid() and CheckSecondValid() routines. These follow a nature parallel pattern as each point needs to be checked against every point in its row and

Algorithm 8.2.1: BUILDSECTIONSBC(*location*)

external $CheckMainValid(), CheckSecondValid()$

global $LatinSquare, UsedPairs$

if location > maxlocation

then We have found a set of sections B and C

Convert location to (x, y) coordinates in square

FOR Every number ∈ S

do
- **if** CheckMainValid(x, y, number) is TRUE
- **then**
  - Place number into LatinSquare(x, y)
  - **for** Every secondnumber ∈ S
  - **do**
    - **if** CheckSecondValid(x, y, secondnumber) is TRUE
    - **then**
      - Place second number into LatinSquare
      - Mark the pair used in UsedPair
      - BuildSectionsBC(location+1)
      - Reset LatinSquare location for secondnumber to zero
      - Unmark the pair in UsedPair
- Reset LatinSquare location for number to zero

column. The parallel machinery of OpenMP can take the for loop and do it automatically. Even this small change is able to increase runtime dramatically as will be shown in Chapter 9.

The added bonus with OpenMP is that it automatically adjusts the processes dependant on the number of processors available. For example, if the loop requires 12 processors and the machine only has 4, it will take three chunks of the problem for each machine. This is, of course, the simplest of the parallel operations.

The next, more complex arrangement is balanced around the protection of data. Clearly in our ISOLS there is alot of dependant data. If we try to work in two places in the same row or column, we can never be sure if one is making the other invalid or vice versa. As well, the matching pairs cannot be placed in their corresponding locations without knowing both what the original value is and what others are used.

There are two key pieces of data that we must protect. These pieces of data are the Latin Square that we are building, and the UsedPair list that we are maintaining. One of these data sets will be protected by the design of the algorithm, the other by the features of OpenMP. We will begin with the design structure.

We already know that we cannot work on a row or column at the same time, so we will work along the diagonal. That allows exclusive access to each row and column during the processing of that point.

Example 8.2.2 Parallel processing locations

7	6	5	4	3	2
Y	X				
	Y	X			
		Y	X		
			Y	X	
				Y	X
					Y

Clearly, we must not progress from the diagonal we are working on to the next until all processors are finished. This is simply the point we set for recursion to happen. All the X values and their corresponding values in section C must be placed prior to the loop starting for Y.

The algorithm alteration is minor. As well, we create an outer loop on the basic algorithm with an advancing (x, y) coordinate corresponding to the processor in which. The location variable now represents the starting point of the diagonal. The OpenMP parallel structure is able to do the rest of the work, dividing the physical processors and the data equally.

One additional update must be made in order to protect the UsedPairs

variable to make sure that its values are safe and useful. This is done by making an update to the CheckSecondValid() routine. The routine is updated so that after running the CheckMainValid(), it enters a critical region defined by OpenMP. A critical region will prevent any other thread from entering that segment. We can check the location without any worry of that data being changed as we work. We also move the marking of the used pair directly into CheckSecondValid(). This is done prior to the end of the critical region, thus the only access to UsedPairs is done during the critical region. The critical region is then ended.

The time that the critical region takes is a very small amount. While it will prevent other processors from continuing, it represents the smallest portion of the processors time. A boolean value needs to be checked in an array, and in some of the cases two booleans set. This proved to be an efficient method for protecting the data and improving run time.

8.2.2 Random Trials

An alternative method that was attempted in order to generate results for the largest of our search attempts (32, 10), was to use randomly generated starting sets.

The basic principle is that we fill the first set of values in the last column of section B.

The settings used for the random trials are broken into three categories. The first is the number of random numbers to chose. These numbers are chosen at the beginning of the algorithm and if the problem backtracks into the randomly chosen number, it is considered a failed path. These locations that are filled are denoted by the X variable in the empty end section shown.

The initial conditions for random start of Latin Square generation

7	6	5	4	3	2
					X
					X
					X
					X
					X

The second value, and the one which required the most experimentation to get an optimal value, is the limiter. Using heuristic methods to control the search, we did bound checking in order to control the search time. In the (32, 10) the bound which results in the deepest searches in the minimum amount of time was 50, 000. It must have advanced at least one position further for each 50, 000 trials it makes. The program tracks all trials made including rebuilding after backtracking so this number can be processed very quickly. As the problem gets further out it is easy to visualize that the trials required to advance increase. By bounding it in this way, paths that can place earlier points easily, thus, are less constricted, and will have more time at the later points and run longer.

The process does not affect the parallel operations. The values are present before the building of sections B and C begins. When that function notices that a value is already contained in the location it was asked to work on, it advances.

The last control variable used is TotalRuns, When a valid solution is not found, it will choose a new set of random values and begin the process again. It essentially operates as a loop. The true power of this randomizing of the search is generated by the bounding function which enhances more probable paths and limits less the probable.

Having generated a set of the sections B and C we are ready to move onto placing the unused pairings which have been carefully maintained in UsedPairs during this step.

8.3 Placing the Missing Pairs

As mentioned previously, the pairs that are not placed in sections B and C must be placed in section A. These pairs will, of course, vary from one possible Latin Square to the next. They can be placed in any location in the Latin Square providing the position does not break any condition of a Latin Square.

When placing the unused pairs we will strictly consider them in pairs. As such, we will deal with a pair of locations in the Latin Square when placing the values, (x, y) and (y, x). If we can place item one of the pair in (x, y) but cannot place item two in (y, x) then it is not a valid location for either item one or two. A solution for this stage of the problem is only found if all missing pairs can be placed within the square. As with the previous stage, this stage must be an exhaustive search as well. Even if we can place a set of points in a location, we still must try all other spots for it as well as all other missing pairs. It is important to note that even with a solution at this point, the majority will not have a complete solution.

The algorithm itself is very similar to building the ends. However, instead of recursively cycling the locations, we will cycle the points to be placed. Prior to running of the algorithm, the UsedPairs structure which holds all the pairings of symbols must be parsed into a list of points. This list will be our index of points to process.

The key values for the algorithm come from the MissingList. It holds the sets of symbols which need to be placed. In the above algorithm, we label these as number and secnumber. Both would be naturally read off of whatever data structure was used to hold MissingList.

The algorithm contains a two level for loop which is necessary to try every

Algorithm 8.3.1: PLACEMISSING(*number*)

```
external CheckMainValid()
global LatinSquare, MissingList
if number > maxnumber
  then We have found an extendable solution

for Every x ≤ size of section A
for Every y ≤ size of section A

do { if LatinSquare(x, y) is empty
       then
     { if CheckMainValid(x, y, number) is TRUE
         then
       { for Every secondnumber ∈ S

           { if CheckMainValid(x, y, secnumber) is TRUE
               then
             do { Place number into LatinSquare
                  Place secondnumber into LatinSquare
                  PlaceMissing(number+1)
                  Reset LatinSquare at this location
```

location in every row and column, which is accomplished by searching every location in a column and then moving on to the next row.

The size of section A is the constraint used for the for loop, which corresponds to the size of the latin square minus the size of the hole, ie, $(20, 6), 20 - 6 = 14$.

The main diagonal of the Latin Square will be skipped each time during the running as the values are preloaded and the first step of the algorithm is a check as to whether the location is empty. Any points placed will also be skipped as each new number set begins from the beginning to ensure exhaustiveness.

The loop itself is very similar to the algorithm used for Building Sections B and C. It makes use of CheckMainValid() twice this time. The use of CheckSecondValid() is to check the UsedPair data, however, in this case it is no longer of concern. Only the basic property of the Latin Square applies. If both of the calls to CheckMainValid() pass then it can add the location to the latin square. The missing pairs do not need to be managed as when the recursion runs, it will automatically advance to the next value to be placed. And lastly, if the path it is on backtracks, the data must be reset in Latin Square.

Checking for a valid solution is done at the start of the cycle by a simple check if there are any more pairs which need to be placed. If there are not, it saves the square and begins looking for other partial solutions it can find in the current square. The algorithm will keep running until all possible solutions from the given square are found. Only then should the next square found in Building Sections B and C be processed.

8.3.1 Parallel Placing the Missing Pairs

Due to the very interconnected nature of section A to sections B and C, it is not possible to process different sets of points in parallel. It is still, of course, possible to add the parallelism mentioned previously to the CheckMainValid(). As well, we can divide the work of checking the pair of points.

When dealing with pairs we have a unique pairing of data, each symbol can only match with it's mate. As well, because they must match with the transpose location they will never be altering data they share as they cannot be on the same row or column.

By using the OpenMP section feature we can process the two instances of CheckMainValid() at the same time. While they appear as a sub loop above they could easily be reversed in order and produce the same fundamental result. Should either fail it is invalid.

A simple change such as dividing the two calls of CheckMainValid() can produce a speed up in the processing, since the processing done in the function is the most intensive portion of the calculation in this stage.

8.4 Finishing the Square

The final step of the procedure is in truth, two steps. Given an extendable solution up until this point, we must fill section A with the set of standard symbols paired with the set of extended symbols. It is clear that the constraint on the standard set is much more strict than that of the extended set. The extended set, up until this point has not been used.

Due to the difference in possible options between the two sets, we will first attempt to place the standard set symbols, following the property of

the Latin Square and ensuring that each symbol appears in every row and column. If properly placed, each symbol in every row must be in every column.

Finding a solution which can place the entire standard set leaves a rather simple task of placing the extended set, ensuring that each symbol matches exactly once with each of the standard set symbols.

In order to fully present this set, it must be done as two separate algorithms. While one in this example does call the second algorithm, it can very easily be implemented to save the information found and have the second step run as a stand alone program. However, if the goal is to simply find a solution, the optimal path, as the second step will usually yield a result, is to solve it immediately, thus, allowing the program to exit with its desired result much faster.

The two algorithms used are BuildSquare, which performs the first step by placing the standard set, and CompleteSquare which places the extended set, thus completing the solution.

In order to make things more concise, the calculation of the point to advance was omitted from the above algorithm. Before an explanation of the algorithm, the method for the calculation will be presented.

Algorithm 8.4.1: BUILDSQUARE($number, y$)

external $CheckMainValid(), CompleteSquare()$

global $LatinSquare$

for Every x $\leq$ size of section A

do
- **if** LatinSquare(x, y) = number)
  - **then**
    - **if** Last number and last row
      - **then** CompleteSquare()
      - **else** $BuildSquare(Advancing)$

for Every x $\leq$ size of section A

do
- **if** LatinSquare(x, y) is empty
  - **then**
    - **if** CheckMainValid(x, y, number) is TRUE
      - **then**
        - LatinSquare(x, y) gets number
        - LatinSquare(y, x) gets -1
        - **if** Last number and last row
          - **then** CompleteSquare()
          - **else** BuildSquare(Advancing)
    - Reset LatinSquare(x, y) and (y, x) to zero

Algorithm 8.4.2: ADVANCE$(number, y)$

external $BuildSquare(number, y)$
global $SizeofSquare, SizeofHole$
if $y > SizeofSquare - SizeofHole$
  then $BuildSquare(number + 1, 1)$
  else $BuildSquare(number, y + 1)$

Advancement in the algorithm is recursive, thus this code could be dropped right into the main algorithm. The program advances the number in the standard set it is currently working on, if it has already placed the symbol in every row. If not, it moves to the next row.

The number and the row are the two condition upon which the algorithm operates. It begins with the number given and the row and first checks if it already exists in the row. CheckMainValid() cannot be used here as only the row is the concern, not whether the square is valid at this step. If it already exists, it must be valid, otherwise it could not have been placed by any of the previous steps, then the algorithm moves on.

Next is the case where the symbol does not exist. The algorithm is run on each and every column in the current row. First, it checks if the location in question is empty. If it contains a value, it will not be altered and moves on to the next square. If the square is empty, it is available for use. If the square is able to accept the number in question, it is placed, otherwise it moves to the next square. When the number is placed one of two things can occur, the square can be complete and call the second algorithm, or the square can advance.

When the numbers are placed, its self orthogonal mate square must be

marked as used, otherwise when the bottom half of the square is searched that square may be filled. To prevent this, rather than leave the symbol 0, to mark unused, we update the square to mark -1 as the unused squares. When this step is complete, all unused squares will contain a -1.

The square is complete if the number placed is the last number, and we are in the last row to be processed. In this case CompleteSquare is called with default parameters to indicate the starting position. Otherwise BuildSquare is run as previously mentioned.

8.4.1 CompleteSquare

Now comes the important step of finding a solution. The final step requires that the extended set of symbols be placed into the Latin Square. This in comparison to the other steps is relatively easy. None of the symbols have been used previously, and there are only enough spaces for them to fit, so there are not a great deal of choices. For example, on the 20, 6 problem, there will be six empty spots that correspond to 1. That leaves only 6 spaces for the first of the extended set, and then 5 and so on. Without many possibilities they can be checked quickly, and because of the lack of constraints, they are easy to place.

The CompleteSquare algorithm works from a starting point of two inputs. A number from the standard set, and a number from the extended set. It works sequentially trying all extended set symbols for the first number from the standard set before advancing to the next number from the standard set. Therefore, the advancement in the recursion is done exactly like that from BuildSquare with just a simple symbol change. As such, it will not be presented again.

Algorithm 8.4.3: COMPLETESQUARE($snum, enum$)

```
external CheckMainValid()
global LatinSquare
for Every x ≤ size of section A
for Every y ≤ size of section A
do { if LatinSquare(x, y) = snum
     then
     { if LatinSquare(y, x) = -1
       then
       { if CheckMainValid(y, x, enum) is TRUE
         then
         { LatinSquare(y, x) gets enum
           if Last snum and last enum
             then A Solution has been found
             else CompleteSquare(Advancing)
         }
         Reset LatinSquare(y, x) to -1
       }
     }
```

The algorithm does a complete search of section A looking for any possible location it can place an enum (Extended Set Member). If it finds a location with a -1, the symbol used to identify empty spots at this stage. Given a matching snum (Standard Set Member) and an empty location, the algorithm can verify that the location is valid for enum. Given the valid location, the symbol can be placed and the algorithm can progress to the next pair of symbols.

Should the last snum and last enum be placed we have a solution. The last snum will always equal the size of section A, and the last enum will always equal the size of Section D. Of course, should a solution not be found, and the algorithm must backtrack, we have the provision in place to reset the data that was updated in this step.

8.4.2 Parallel Finishing the Square

Just as in the Place Missing algorithm, the possibilities for parallel application are somewhat limited. Due to every location in section A being dependent the square cannot be internally divided to be computed in parallel. As with all previous cases CheckMainValid() can be utilized to improve performance, however, due to the relatively long runtime of BuildSquare, the most efficient method for parallel is to subdivide the list of squares.

It is a simple task within the OpenMP environment to split off a loop into its own section. The program is simply entered into a parallel loop at the beginning of its main loop, thus each processor is responsible for first reading in a set of squares and then processing them. This coarse grain of parallelism works very well when processing each step is very time consuming as it is in this algorithm.

That being said, a balance must be reached between the processor usage and the memory. We found the best method was to establish a block of memory that could easily be contained in memory and turn it into a private memory block. Each processor then protects its own set of data and then loads the next block when it runs out of data. Reading and writing of files is the only necessary critical region in this process and can be implemented within the individual functions.

Providing each processor does not take more of a block than the memory can handle, with taking into account how the problem grows, this method can greatly optimize run time. The same could be accomplished by creating many instances of the program using separate file names, and input files, however, the extra loading and accessing of common routines increases the time. As well, keeping all the individual files open makes the result less than optimal.

This concludes the methods used in the search for ISOLS. The methods detailed in this chapter can be implemented to solve any ISOLS, the methods used to optimize the search can be implemented for a Latin Square of any size.

Chapter 9

Summary

In presenting the results of this research, we will use the system described in chapter 7 as a speed reference in our search. SATO's ability to solve the ISOLS(14,4) problem makes it ideal to match against our specific solver application.

For all comparisons, the program was run on our available parallel machine. As such, we are provided with the ability to reference the speed of execution of the parallel versions of the program against their equivalent sequential versions. Through these comparisons we will show a large improvement in the reduction of search time.

Our ultimate goal, of course, was to solve for the three open cases of ISOLS. At this time, we will present the current state of the problem and our relative distance to the solution.

9.1 Speed Comparisons

The comparison on speed will be our primary evidence as to the effectiveness of our implementation. The primary comparison will be in comparing the two methods, that of our targeted approach and the general solver method used by SATO. While the method developed by SATO can be used to solve many problems, it will be shown that the specific solver is able to improve on performance by using the alternate structure to subdivide the problem.

In order to fairly calculate the performance of both algorithms, the source code for SATO was downloaded and compiled on our local parallel machine. The on site parallel machine is a forty processor SGI Origin 2000, with a total memory of sixteen gigabytes. Both programs were run at separate times under similar load conditions on the machine. Due to the size of the ISOLS(14,4) problem we used it as a base comparison as its running time was reasonable enough to execute a number of times and compare the results.

One small difference should be noted between the results, our implementation presets the ISOLS into standard form, whereas the SATO application does not. Both are still able to produce a valid result.

Comparing the sequential version of our solution to that of SATO provides interesting values, the total time of solution in SATO for the (14,4) problem was 29 days, 7 hours, 40 minutes and 5 seconds. Timing is referenced by the unix time function in c and can be considered accurate to within 24 ms. Thus, by comparing only to the second we can assume the value to be correct. To compare the above times to our results, the total run time of our specific program on the (14,4) set was 5 days, 9 hours, 25 minutes, 28 seconds.

The time difference corresponds to an improvement of almost 600 percent, by running almost six times faster it can be considered a significant improve-

ment. As well, this is the comparison using strictly sequential running. The running time is further reduced by adding in the parallel optimizations.

In comparing the parallel, we see the greatest speed improvements in the first and last step. This test was done using two different methods. The first approach used was to run it using only general parallelism in the CheckMainValid() routine using the OpenMP for loop construct. This would be our baseline for parallel, while the second would provide our ideal optimizations.

Through the trials it became clear that the place missing operation is only able to provide a small improvement. Our second implementation method is to activate all presented parallel methods in chapter 8 and for the last portion the code is updated so that each processor takes a set of data as mentioned with each processor taking one hundred possible squares at a time.

Upon running it under these conditions the improvement of the program under parallel was immediately evident. The total run time for the simple parallel setup was 3 days 18 hours, 23 minutes 34 seconds. The total run time for the (14,4) with all parallel optimizations became, 2 days, 3 hours, 48 minutes, 13 seconds. The basic parallelism represents a thirty percent increase in speed while the fully optimized version is sixty percent more efficient that the same sequential application. With the extra processing needed to handle the parallel application, this can be considered a very large improvement by having a good balance of the work performed between the processors.

Our targeted approach was able to provide a significant improvement in performance in both its sequential modes and parallel modes. By eliminating many cases in our subdivided approach the total search space was greatly reduced.

The resulting Latin Square as generated by SATO is:

1	7	λ	8	γ	4	9	θ	10	α	6	5	3	2
θ	2	8	γ	6	α	3	5	λ	4	10	7	9	1
9	α	3	10	8	λ	θ	7	γ	2	5	6	1	4
λ	8	θ	4	7	5	10	2	α	γ	9	1	6	3
10	γ	1	α	5	7	4	3	θ	λ	8	9	2	6
7	9	2	θ	λ	6	γ	α	8	3	1	4	10	5
γ	10	9	λ	α	1	7	6	4	θ	3	2	5	8
6	λ	γ	9	θ	3	α	8	5	1	2	10	4	7
α	6	4	1	2	θ	λ	γ	9	5	7	3	8	10
5	θ	α	2	3	γ	1	λ	6	10	4	8	7	9
8	1	7	3	10	9	5	4	2	6				
2	3	10	5	4	8	6	9	1	7				
4	5	6	7	1	2	8	10	3	9				
3	4	5	6	9	10	2	1	7	8				

The resulting Latin Square generated by our application is:

1	7	λ	α	6	8	9	θ	10	γ	5	4	3	2
θ	2	8	5	γ	α	3	4	λ	6	10	7	9	1
9	γ	3	8	λ	10	θ	7	α	2	4	5	1	6
10	α	1	4	7	γ	6	3	θ	λ	8	9	2	5
7	9	2	λ	5	θ	α	γ	8	3	1	6	10	4
λ	8	θ	7	4	6	10	2	γ	α	9	1	5	3
α	10	9	γ	1	λ	7	5	6	θ	3	2	4	8
5	λ	α	θ	3	9	γ	8	4	1	2	10	6	7
γ	5	6	2	θ	1	λ	α	9	4	7	3	8	10
4	θ	γ	3	α	2	1	λ	5	10	6	8	7	9
8	1	7	10	9	3	4	6	2	5				
2	3	10	6	8	4	5	9	1	7				
6	4	5	1	2	7	8	10	3	9				
3	6	4	9	10	5	2	1	7	8				

9.2 Searching for ISOLS(20,6)

As mentioned the focus of this paper was initially to find ISOLS(20,6). After finding a successful solution to (14,4) this appeared to be an emminent possibility. However it soon became clear that while (20, 6) is a larger problem, it is not of the scale that was expected. Clearly, any processing with (20, 6) will be of a greater magnitude. The true processing difference came with the number of possible latin squares it was able to find.

In the (14,4) problem, the total Squares comprised of sections B and C generated which needed to be processed was 3521. Processing these took approximately forty five seconds each. Thus, once the first step is complete we can proceed to trying all the options and have a solution within 44 hours or 2 days. Naturally, in most cases, the full number will never need to be searched. With the ISOLS(20, 6) problem we came across a different case. After running the program for over six weeks, no solution had been found. To better gauge the time frame we would be looking at, the first step was executed in isolation, in order to generate all possible results from the first step before continuing. Without knowing where the ISOLS can be found in relation to the list of the initial Squares we must assume that the exhaustive search will be necessary. Thus, the following data results.

During the first stage, the total number of squares generated for the (20, 6) problem which needed to be processed equaled over 58 million. 58 192 203 to be exact. Clearly this problem is very large and will take a large time to solve. The average time to complete a square for this problem is five minutes and fifty one seconds. This results in a total run time using the existing methods to be 1201 years, 97 days, 5 hours, 6 minutes, 4 seconds. While it is able to run faster than any previous method, it is still not feasible to run

within an acceptable time frame.

Another alternative was to look at the equivalent cases within the building of sections B and C, which has the ability to greatly reduce the number of valid cases. However, the maximum possible improvement would be a reduction on the scale of 1:21. Thus, reducing the search time to a total of 57.5 years. Even if this update were implemented, the size of the search remains too large to currently be solved.

While the total number of open cases was not reduced within this paper, the volume of possible solutions leads to a strong belief that a solution to ISOLS(20, 6) does exist. However, even without locating a new solution, the improvements made to the search method are very evident as shown by the results of the previous section. It would be wise for any new algorithm to begin with a subdivision of the problem as does here, and if possible, utilize an appropriate parallel structure.

9.3 Searching for ISOLS(26, 8), (32, 10)

In searching for the open cases of ISOLS, we focused upon the smallest of the problems the (20, 6) however, time was spent attempting to locate the larger two cases as well.

In searching for (26, 8), there was early success as it was able to find a number of solutions for the first two stages of the problem, however none of these possible solutions lead to a final solution. Due to time constraints after eight weeks of running this problem we reallocated the processor time to focus on the (20, 6) problem.

The same approach was taken to the (32, 10) problem. During the eight

weeks it was allowed to run however, it was not able to generate even an introductory solution. The size of the hole created sections B and C that were sufficiently large enough to make finding a solution a large task. In the attempts to find a valid solution which would allow us to continue the search for (32,10) we tried some alternative method. As mentioned in the previous chapter we attempted the random building method, however, during the four weeks that method was used, no result could be generated either.

While these larger problems did not provide us with the details to the nature of the problem as specifically as we were able to gain from the (20, 6) problem, it did become clear that the problem is very large, and some special construction beyond what we have developed thus far will be necessary to prove the existence of these open cases.

Bibliography

[Abel] Abel, R., Colbourn, Charles J., Dinitz, Jeffrey H., *Incomplete MOLS* in: CRC Handbook of Combinatorial Designs, CRC Press. C.J. Colbourn and J. H. Dinitz edit. Boca Raton FL, (1996)

[Bennett] Bennett, F.E., Abel, R., Zhang, H., Zhu, L., *A few more incomplete self-orthogonal Latin squares and related designs* in: Australasian J. Combin. 21 (2000), 85-94.

[Berman] Berman, Kenneth A., Paul, Jerome L., (1997). *Sequential and Parallel Algorithms* PWS Publishing Company.

[Bierbrauer] Bierbrauer, J., Colbourn, Charles J., *Orthgonal Arrays of Strength More than Two* in: CRC Handbook of Combinatorial Designs, CRC Press. C.J. Colbourn and J. H. Dinitz edit. Boca Raton FL, (1996)

[Brayton] Brayton, R.K., Coppersmith, D. Hoffmam, A.J., (1976). *Self-orthogonal latin squares, Teorie Combinatorie, Proceedings of the Rome Conference*

[Brouwer] Brouwer, Andries, Abel, R., Colbourn, Charles J., Dinitz, Jeffrey H., *Mutually Orthogonal Latin Squares* in: CRC Handbook

of Combinatorial Designs, CRC Press. C.J. Colbourn and J. H. Dinitz edit. Boca Raton FL, (1996)

[Bruck] Bruck, R. H., Ryser, H. J., (1949) *The non-existence of certain finite projective planes, Can. J. Math., Vol. 1*

[Colbourn] Colbourn, Charles J., Dinitz, Jeffrey H., *Latin Squares*

[Euler] Euler, L., (1782). *Recherches sur une nouvelle espce de quarrs magiques Verh. Zeeuwsch. Genootsch. Wetensch. Vlissengen Vol. 9*

[Gopalakrishnan] Gopalakrishnan, K., Stinson, Douglas R., *Applications of Designs to Cryptography* in: CRC Handbook of Combinatorial Designs, CRC Press. C.J. Colbourn and J. H. Dinitz edit. Boca Raton FL, (1996)

[Graham Vol 1] Graham, R.L., Grotschel, M., Lovasz, L., (1995) *Handbook of Combinatorics Volume 1* The MIT Press Cambridge, Massachusetts.

[Graham Vol 2] Graham, R.L., Grotschel, M., Lovasz, L., (1995) *Handbook of Combinatorics Volume 2* The MIT Press Cambridge, Massachusetts.

[Hall] Hall, Marshall, *Combinatorial Theory: Chapter 13*

[Kreher] Kreher, Donald L., Stinson, Douglas R., (1999). *Combinatorial Algorithms: Chapters 4 and 5.* CRC Press LLC.

[Liu] Liu, C. L., (1968) *Introduction to Combinatorial Mathematics* McGraw-Hill Inc.

[Stickel] Stickel, Mark E., Zhang, Hantao *Implementing the Davis-Putnam Algorithm by Tries*

[Stinson] Stinson, D.R., (2002) *Combinatorial Designs: Constructions and Analysis*

[Street] Street, Anne P., Wallis, W. D., (1977) *Combitorial Theory: An Introduction* CBRC, Winnipeg, Canada.

[Tarry] Tarry, G., (1900) *Le problme des 36 officiers, C. R. Assoc. Fran. Av. Sci. Vol. 1 and Vol. 2 (1901)*

[Wei] Wei, R., Stinson, D. R., (1998) *Combinatorial Properties and constructions of traceability schemes and frameproof codes* Society for Industrial and Applied Mathematics.

[Xu] Xu, Y., Zhang, H., Zhu, L., (2002) *Existence of frame SOLS of type* $a^n b^1$

[Zhang] Zhang, Hantao, *Specifying Latin Square Problems in Propositional Logic*

[Zhang 2] Zhang, Hantao *SATO: an Efficient Propositional Prover*

[Zhu] Zhu, L., *Self Orthogonal Latin Squares* in: CRC Handbook of Combinatorial Designs, CRC Press. C.J. Colbourn and J. H. Dinitz edit. Boca Raton FL, (1996)

Index